26° SIMPÓSIO BRASILEIRO DE REDES DE COMPUTADORES E SISTEMAS DISTRIBUÍDOS

26 a 30 de Maio de 2008
Rio de Janeiro, RJ

IX WORKSHOP DE TESTES E TOLERÂNCIA A FALHAS

ANAIS

Organizadores
Raul Ceretta Nunes
... (SBRC)

Realização
Laboratório Nacional de Computação Científica – LNCC
Universidade Federal Fluminense – UFF
Universidade Federal do Rio de Janeiro – UFRJ
Universidade Estadual do Rio de Janeiro – UERJ

Promoção
Sociedade Brasileira de Computação – SBC
Laboratório Nacional de Redes de Computadores – LARC

Coordenador do Comitê de Programa do WTF
Raul Ceretta Nunes (UFSM)

Comitê de Programa do WTF 2008
Alcides Calsavara (PUCPR)
Alexandre Sztajnberg (UERJ)
Alysson Bessani (Universidade de Lisboa)
Ana Maria Ambrosio (INPE)
Avelino Zorzo (PUCRS)
Cecilia Rubira (UNICAMP)
Edmundo Madeira (UNICAMP)
Eliane Martins (UNICAMP)
Elias P. Duarte Jr. (UFPR)
Fabíola Greve (UFBA)
Fernando Dotti (PUCRS)
Fernando Pedoni (University of Lugano)
Flávio Assis Silva (UFBA)
Francisco Brasileiro (UFCG)
Francisco José Silva (UFMA)
Frank Siqueira (UFSC)
George Lima (UFBA)
Henrique Madeira (Universidade de Coimbra)
Ingrid Jansch-Pôrto (UFRGS)
Irineu Sotoma (UFMS)
Joni da Silva Fraga (UFSC)
Jose Pereira (Universidade do Minho)
Lau Cheuk Lung (UFSC)
Luiz Carlos Albini (UFPR)
Luiz Eduardo Buzato (UNICAMP)
Luiz Nacamura Júnior (UTFPR)
Marinho Barcellos (PUCRS)
Miguel Correia (Universidade de Lisboa)
Noemi Rodriguez (PUC-Rio)
Orlando Loques (UFF)
Patricia Machado (UFCG)
Raimundo Barreto (UFAM)
Raimundo José de Araújo Macêdo (UFBA)
Regina Moraes (UNICAMP)
Rogerio de Lemos (University of Kent)
Rui Oliveira (Universidade do Minho)
Sérgio Gorender (UFBA)
Silvia Vergilio (UFPR)
Taisy Weber (UFRGS)
Thais Vasconcelos Batista (UFRN)
Udo Fritzke Jr. (PUCMG - Poços de Caldas)
Vidal Martins (PUCPR)

Revisores do WTF 2008
Alan Nakai (UNICAMP)
Alcides Calsavara (PUCPR)
Alexandre Sztajnberg (UERJ)
Alysson Bessani (Universidade de Lisboa)
Ana Maria Ambrosio (INPE)
Avelino Zorzo (PUCRS)
Cecilia Rubira (UNICAMP)
Claudio Carvalho (UNICAMP)
Daniela Barreiro Claro (UFBA)
Eliane Martins (UNICAMP)
Elias P. Duarte Jr. (UFPR)
Emerson Mello (UFSC)
Fabíola Greve (UFBA)
Fernando Dotti (PUCRS)
Flávio Assis Silva (UFBA)
Francisco Brasileiro (UFCG)
Francisco José Silva (UFMA)
Frank Siqueira (UFSC)
George Lima (UFBA)
Henrique Madeira (Universidade de Coimbra)
Ingrid Jansch-Pôrto (UFRGS)
Irineu Sotoma (UFMS)
Jose Pereira (Universidade do Minho)
Lasaro Camargos (University of Lugano)
Lau Cheuk Lung (UFSC)
Luiz Carlos Albini (UFPR)
Luiz Eduardo Buzato (UNICAMP)
Luiz Nacamura Júnior (UTFPR)
Makelli Jucá (UFCG)
Marcos Madruga (UFRN)
Marinho Barcellos (PUCRS)
Miguel Correia (Universidade de Lisboa)
Noemi Rodriguez (PUC-Rio)
Orlando Loques (UFF)
Patricia Machado (UFCG)
Paul Regnier (UFBA)
Raimundo Barreto (UFAM)
Raimundo José de Araújo Macêdo (UFBA)
Regina Moraes (UNICAMP)
Roberto Bittencourt (UEFS)
Rogerio de Lemos (University of Kent)
Rui Oliveira (Universidade do Minho)
Sérgio Gorender (UFBA)
Silvia Vergilio (UFPR)
Taisy Weber (UFRGS)
Thais Vasconcelos Batista (UFRN)
Udo Fritzke Jr. (PUCMG - Poços de Caldas)
Vidal Martins (PUCPR)

Promoção

Sociedade Brasileira de Computação – SBC

Presidente
José Carlos Maldonado (ICMC - USP)

Vice-Presidente
Virgílio Augusto Fernandes Almeida (UFMG)

Diretorias:

Administrativa
Carla Maria Dal Sasso Freitas (UFRGS)

Finanças
Paulo Cesar Masiero (ICMC - USP)

Eventos e Comissões Especiais
Marcelo Walter (UFPE)

Educação
Edson Norberto Cáceres (UFMS)

Publicações
Karin Breitman (PUC-Rio)

Planejamento e Programas Especiais
Augusto Sampaio (UFPE)

Secretarias Regionais
Aline Maria Santos Andrade (UFBA)

Divulgação e Marketing
Altigran Soares da Silva (UFAM)

Diretorias Extraordinárias:

Regulamentação da Profissão
Ricardo de Oliveira Anido (UNICAMP)

Eventos Especiais
Carlos Eduardo Ferreira (USP)

Cooperação com Sociedades Científicas
Taisy Silva Weber (UFRGS)

Conselho:

Mandato 2007-2011
Cláudia Maria Bauzer Medeiros (UNICAMP)
Roberto da Silva Bigonha (UFMG)
Cláudio Leonardo Lucchesi (UNICAMP)
Daltro José Nunes (UFRGS)
André Ponce de Leon F. de Carvalho (ICMC - USP)

Mandato 2005 - 2009
Ana Carolina Salgado (UFPE)
Jaime Simão Sichman (USP)
Daniel Schwabe (PUC-Rio)
Vera Lúcia Strube de Lima (PUCRS)
Raul Sidnei Wazlawick (UFSC)

Suplentes - Mandato 2007-2009
Ricardo Augusto da Luz Reis (UFRGS)
Jacques Wainer (UNICAMP)
Marta Lima de Queiroz Mattoso (UFRJ)

Laboratório Nacional de Redes de Computadores - LARC

Diretora do Conselho Técnico-Científico:
Luci Pirmez (UFRJ)

Vice-Diretora do Conselho Técnico-Científico:
Thais Vasconcelos Batista (UFRN)

Diretora Executiva:
Flávia Coimbra Delicato (UFRN)

Vice-Diretor Executivo:
Artur Ziviani (LNCC)

Realização

Coordenação Geral do SBRC
Artur Ziviani (LNCC)
Célio Vinicius Neves de Albuquerque (UFF)
Luís Henrique Maciel Kosmalski Costa (UFRJ)
Marcelo Gonçalves Rubinstein (UERJ)

Coordenador do Comitê de Programa
Nelson Luís S. da Fonseca (Unicamp)

Coordenador de Tutoriais
Otto Carlos M. B. Duarte (UFRJ)

Coordenador de Minicursos
Luciano P. Gaspary (UFRGS)

Coordenador de Painéis e Debates
Luiz Fernando G. Soares (PUC-Rio)

Coordenador do Salão de Ferramentas
Flávia Coimbra Delicato (UFRN)

Coordenador de Workshops
Alexandre Sztajnberg (UERJ)

Comitê Consultivo
Antônio Jorge Gomes Abelém (UFPA)
Carlos Alberto Maziero (PUC-PR)
Elias Procópio Duarte Jr. (UFPR)
Keiko Verônica Ono Fonseca (UTFPR)
João Crisóstomo Weyl Costa (UFPA)
Joni da Silva Fraga (UFSC)
Lisandro Zambenedetti Granville (UFRGS)
Luci Pirmez (UFRJ)

Assessoria do Comitê de Programa
André Costa Drummond (Unicamp)
Daniel Macedo Batista (Unicamp**)**

Webmaster
Jairo Duarte (UFF)
Luís Henrique M. K. Costa (UFRJ)

Prefácio

O Workshop de Testes e Tolerância a Falhas (WTF) é um evento anual promovido pela Comissão Especial de Sistemas Tolerante a Falhas da Sociedade Brasileira de Computação (SBC) e tem como objetivo propiciar a integração entre pesquisadores da áreas de Tolerância a Falhas e Testes & Validação, bem como o fomento da pesquisa por técnicas para a construção de sistemas computacionais confiáveis e de alta disponibilidade. Desde 2003, o WTF é realizado em conjunto com o Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC), maior evento brasileiro que tem forte correlação com os temas cobertos no WTF, fomentando assim também a integração com pesquisadores de áreas correlatas.

Em sua nona edição (2008), o WTF contou com 29 submissões de artigos completos, dos quais 14 foram selecionados para publicação e apresentação (48,27%). A exemplo das outras edições, o Comitê de Programa foi constituído por brasileiros e portugueses, num total de 42 pesquisadores entre brasileiros e portugueses. O Comitê de Programa também contou com o apoio de 10 avaliadores externos. Das submissões observou-se que o evento despertou o interesse de 72 autores brasileiros, 5 portugueses e 1 espanhol. Tais dados indicam a consolidação da integração entre as comunidades científicas brasileira e portuguesa e a clara importância deste evento em sua manutenção. Em especial, agradeço a imensurável contribuição de todos os membros do comitê de programa e revisores pela competência e dedicação na condução do processo de avaliação dos artigos.

A programação do evento está composta por sete sessões técnicas (Automação de Testes; Injeção de Falhas; Ferramentas de Testes; Algoritmos Distribuídos I; Algoritmos Distribuídos II; Protocolos de Comunicação; e Reconfiguração em Middlewares) e uma palestra nacional (artigo convidado) intitulada “Caracterização de Falhas Relacionadas a Aplicações de *Live Streaming* na Internet”, a ser ministrada pela pesquisadora Ingrid Jansch-Pôrto da UFRGS. Com o intuito de promover o debate cada sessão técnica está composta por apenas dois artigos no mesmo tema e um espaço único para perguntas e respostas.

Gostaria de aproveitar a oportunidade para agradecer também os organizadores do SBRC 2008, em especial o Coordenador de Workshops Alexandre Sztajnberg (UERJ), pelo constante apoio dedicado a esta comunidade científica e saudar todos os participantes do IX Workshop de Testes e Tolerância a Falhas.

Um ótimo workshop a todos!

Santa Maria, maio de 2008.

Raul Ceretta Nunes
Coordenador do Comitê de Programa do WTF 2008

Sumário

Sessão Técnica 07 – Reconfiguração em Middlewares

Índice de Autores

Caracterização de Falhas Relacionadas a Aplicações de *Live Streaming* na Internet

Ingrid Jansch-Pôrto

Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

ingrid@inf.ufrgs.br

Abstract. *Live streaming multicast applications that use Internet support – and especially those that use a peer-to-peer overlay – are subject to natural and human faults. In order to survive and to be attractive from the point-of-view of its (many) thousands of users, many management mechanisms must be implemented. This paper aims at presenting the main fault types that affect this kind of systems. Some comments on fault modeling, and about problems and reasons related to handling these faults are added. This paper is not supposed to offer solutions, but intends to give insights to novice researchers in this domain.*

Resumo. *Sistemas de difusão de conteúdos em tempo real – live streaming – construídos sobre a Internet, e especialmente aqueles que exploram arquiteturas peer-to peer, estão sujeitos a falhas naturais e humanas. Para que as aplicações não só sobrevivam, mas apresentem características atrativas aos (muitos) milhares de usuários, diversos mecanismos precisam ser usados como suporte. Este artigo visa apresentar os principais tipos de falhas que afetam estes sistemas e bem como tecer comentários sobre dificuldades relacionadas ao seu tratamento, além de informações sobre a modelagem de falhas. O artigo não pretende oferecer soluções, mas servir como um alerta a pesquisadores iniciantes na área.*

1. Introdução

Atualmente, a difusão de vídeo e áudio ocupa um alto percentual dos conteúdos que são do interesse de usuários da Internet. No Brasil, a forma mais conhecida ainda é a "sob demanda", na qual os usuários buscam arquivos pré-armazenados. Neste caso, o ritmo da operação de transferência depende das características de origem e destino – tais como as capacidades de *download* e *upload*; do momento de busca – quando pode haver competição entre usuários que tentam ter acesso simultaneamente a conteúdos recém disponibilizados, sobrecarregando o servidor; ou estarem disponíveis apenas em raros usuários cujos recursos de acesso à rede são modestos. Estes arquivos podem estar subdivididos em partes, coletáveis em qualquer ordem e depois remontadas. Apesar da ansiedade por poder desfrutar dos arquivos o mais cedo possível, de forma geral, é apenas após a conclusão da transferência integral do arquivo que o usuário verá o filme ou vídeo, escutará a música ou assistirá o programa, de acordo com seu ritmo ou tempo livre. Dependendo de várias condições, esse processo de transferência pode exigir horas ou dias para ser concluído.

Por outro lado, vem crescendo o escopo de aplicações nas quais o usuário tem interesse em programas "ao vivo". Situações tais como eventos esportivos, debates

políticos, conferências ou palestras seguidas de perguntas/respostas são exemplos dessas aplicações, denominadas na literatura como *live streaming* (difusão ao vivo). Nesse caso, as informações são geradas exatamente antes da sua transmissão (não estão disponíveis previamente) e disseminadas em tempo real. Para que se mantenham o interesse e a satisfação dos usuários, é necessário que elas sejam recebidas por eles com atraso mínimo e de forma regular (compassada) para manter a continuidade de exibição.

A tecnologia *peer-to-peer* (P2P) aparece como um suporte interessante para este tipo de transmissão: os participantes colaboram na redistribuição de conteúdos, aproveitando recursos disponíveis na rede e multiplicando a capacidade de difusão à medida que se multiplicam os usuários interessados, contribuindo na escalabilidade. Assim, a multiplicação dessas aplicações de *live streaming* tem ocorrido de forma rápida em países onde a Internet foi implantada recentemente, com moderna tecnologia, como na China: lá, canais de televisão vêm sendo transmitidos a milhares de usuários através dessa tecnologia (http://www.pplive.com). Em outros lugares, como nos EUA, vem crescendo com a migração maciça de usuários que abandonam a rede discada e filiam-se a provedores de conexões em banda larga.

Entretanto, as características abertas da Internet fazem com que algumas dificuldades surjam nesses tipos de aplicações: a liberdade de reunir-se à programação ou desvincular-se a qualquer momento – por perda de conexão ou de interesse – são exemplos. Eles causam impactos sobre parte dos demais nodos (*peers*). Embora o paradigma P2P favoreça a escalabilidade dos sistemas com o crescimento do número de usuários, ele também os deixa vulneráveis a comportamentos oportunísticos. Nodos oportunistas tentam receber o fluxo de dados sem retribuir de forma correspondente, reduzindo assim a capacidade total de *upload* do sistema [Haridasan, Jansch-Pôrto e van Renesse 2008]. Isso pode ocorrer, por exemplo, quando eles não contribuem com a propagação de pacotes – recebem-nos, mas não os retransmitem ou retransmitem preguiçosamente (aquém de sua capacidade ou do que é prescrito pelo protocolo de operação). Além disso, a anonimidade encoraja parte dos usuários a tentar fraudar regras de participação. E ainda o próprio meio de comunicação é sucetível a ruídos, os quais causam corrupção de pacotes, atrasos e desconexões. Essas e outras situações causam problemas nos sistemas de *live streaming* que empregam a Internet como suporte natural para a difusão de informações em tempo real. Logo, resiliência é um requisito-chave para *live streaming* em arquiteturas *peer-to-peer* [Fodor e Dán 2007].

A partir do contexto descrito, o objetivo do presente artigo é o de caracterizar comportamentos anômalos durante o período operacional nesse tipo de aplicações. Vários termos referentes ao comportamento dos diversos perfis de usuários aparecem na literatura: nodos corretos [Castro et al. 2002] [Singh et al. 2004], altruístas [DaSilva e Srivastava 2004], egoístas (do inglês: *selfish* [Hales 2004]), *freeloaders* [Ge et al. 2003], racionais [Shneidman e Parkes 2003], bizantinos [Awerbuch et al. 2002] [Aiyer et al. 2005] [Haridasan e van Renesse 2006] – eles serão estudados aqui, em conjunto com as formas mais conhecidas de burla aos sistemas. Ao longo dessa apresentação, serão feitas algumas reflexões sobre como lidar com esses problemas ou referentes aos desafios que eles apresentam em face do contexto de trabalho.

Logo após esta introdução, seguem informações referentes às aplicações de *live streaming* para fornecer subsídios aos leitores não iniciados na área (Seção 2). Em seguida, é revisada a terminologia usada na área de falhas (Seção 3). Propõe-se então

uma taxonomia para as falhas que refletem o comportamento dos usuários (Subseção 3.2). A Seção 4 explica como são vistas falhas num contexto de *live streaming*, e apresentam-se os perfis de desvios básicos de usuários (não intencionais na Subseção 4.1), com ênfase nas ocorrências intencionais (Subseções 4.2 e 4.3). Comentários sobre a modelagem de falhas completam o conteúdo do artigo (Seção 5), seguido apenas pelo fechamento na Seção 6, de conclusões.

2. Características das aplicações *live streaming*

Nas aplicações de *live streaming*, os conteúdos são gerados "ao vivo" e são então distribuídos para os usuários. Em geral, estes conteúdos são subdivididos em pequenos pacotes com identificação de ordem e codificação, o que permite recuperação de erros limitada. Os usuários vão coletando os pacotes e exibem-nos à medida em que a seqüência de pacotes vai se completando (*playback*). Pacotes excessivamente atrasados não fazem mais sentido aos usuários, pois a seqüência na qual eles seriam inseridos provavelmente já foi exibida – assim, são descartáveis. A quantidade de pacotes recebidos com relação ao número de pacotes enviados, que é dado por um percentual, é dito **índice de continuidade** (*continuity index*). A partir do valor deste percentual pode-se avaliar a qualidade das informações recebidas pelos usuários; entretanto, é necessário ressaltar que este percentual pode variar significativamente entre diferentes usuários em uma rede P2P.

Essas aplicações (*live streaming multicast*), através das quais conteúdos ao vivo são difundidos através de Internet, podem ser baseadas em uma infraestrutura de equipamentos e serviços oferecida pelo provedor do serviço ou em redes P2P. No primeiro (infraestrutura), existe a necessidade de posicionar, em locais estratégicos na Internet, servidores de vídeo e nodos de difusão no nível da aplicação, de tal forma que o vídeo seja transmitido pelos servidores aos nodos especiais e, então, aos clientes. Além de ser cara e exigir considerável esforço de manutenção, esta opção dificilmente permite atender um grupo potencialmente grande (na ordem de milhares) de usuários simultâneos, em virtude da enorme demanda de capacidade de processamento, armazenamento e comunicação exigida de poucos nodos de difusão. Aplicações de difusão que adotam tal estratégia, como as populares YouTube e Yahoo! Video, lidam com tal dificuldade, limitando drasticamente o tamanho dos conteúdos disseminados.

Por outro lado, nas redes *peer-to-peer*, onde os nodos estão conectados através de uma rede lógica (*overlay*), a distribuição não se apoia sobre nodos dedicados: cada nodo atua no recebimento da informação e na sua redistribuição a outros, através da exploração cooperativa da capacidade de *upload* dos nodos. É importante ressaltar que a escolha de uma solução centralizada ou distribuída requer dos projetistas dos sistemas uma avaliação cuidadosa da expectativa de demanda a fim de que possam prover uma aplicação que satisfaça a aspectos como qualidade da informação recebida pelo usuário e desempenho, com baixos atrasos. A quantidade de nodos que participam do *overlay* pode mudar rapidamente; os pacotes precisam ser transmitidos com atrasos fim-a-fim aceitáveis para aplicações "ao vivo", que é em torno de dezenas de segundos; e ainda, para manter aceitável a qualidade do vídeo percebida, a taxa de perda de pacotes precisa ser baixa [Fodor e Dán, 2007].

As duas formas predominantes de organização dos *overlays* são: a) em árvores ou *push-based streaming*; b) organizadas randômicamente, também referidas como *mesh*, *pull-based streaming* ou *data-driven randomized*. Encontram-se ainda

combinações das anteriores: árvores múltiplas ou árvores combinadas com *mesh*, por exemplo. Observe-se que não há como empregar uma única árvore para organizar os nodos pensando em assegurar qualquer nível de resiliência.

Com o uso de árvores, os dados do *streaming* são repassados de pais para filhos, tendo o servidor como raiz. A alternativa de múltiplas árvores tem sido usada: a) para resolver o desbalanceamento de contribuição de nodos internos à árvore e os nodos-folhas; b) para garantir a continuidade de distribuição de pacotes quando nodos internos abandonam a rede; c) para aproveitar a disponibilidade de largura de banda de nodos internos. Dependendo dos critérios usados na escolha dos nodos para a composição das múltiplas árvores, a manutenção pode não ser simples. Além disso, a exploração de uma estrutura definida deixa esta topologia mais suscetível a ataques.

Abordagens que exploram estruturas randômicas, definidas ao longo de sua construção principalmente com base no número médio de vizinhos, distribuem os pacotes através de um método *pull* (ou *swarming*). Os nodos trocam mensagens de controle com seus vizinhos para divulgar os conteúdos disponíveis e receber requisições desses; esse protocolo evita o recebimento de pacotes duplicados de vizinhos diferentes, mas introduz algum atraso (embora ele afete de forma quase regular a todo o conteúdo). Devido à multiplicidade de vizinhos, a estrutura tende a ser resiliente ao comportamento dinâmico dos nodos, pois mantém-se a troca de pacotes enquanto ocorre a manutenção do *overlay*. Sua característica randômica também a torna mais robusta a ataques.

Hoje, há diversos sistemas práticos que permitem o acesso a um grande número de usuários interessados em receber fluxo de dados em tempo real, sem requisitarem quantidade extensiva de recursos. Sistemas recentes tais como Chainsaw [Pai, Kumar, Kamilmani, Sambamurthy e Mohr 2005] e Coolstreaming [Zhang, Liu, Li e Yum 2005] mostraram que o uso de nodos organizados através de um *overlay* de composição randômica (*mesh*) e disseminação de dados *pull-based* pode proporcionar uma boa resiliência a falhas e *churn* [Haridasan e van Renesse 2006], [Magharei e Rejaie 2007]. ***Churn*** é o termo usado para expressar situações nas quais ocorrem múltiplas saídas de nodos participantes simultaneamente.

3. Tipos de falhas

O sistema aqui considerado é composto por um conjunto numeroso de máquinas, ou nodos, que correspondem a usuários interessados nos serviços disponíveis, conectadas à Internet. A tecnologia de conexão é baseada no paradigma *peer-to-peer* (P2P): os nodos estão interessados em receber conteúdos e por isso se dispõem também a disseminá-los entre os demais participantes, cooperando na distribuição e, portanto, reduzindo a quantidade de trabalho do servidor principal ou *source* (fonte de dados, único, dedicado, em geral; também denominado *seed* ou *root*). A comunicação entre participantes é definida através de uma rede *overlay*, que pode explorar diferentes formas de conexão (e.g., árvore, *mesh* ou combinações dessas). A escolha do paradigma P2P, assim como as características da Internet, fazem com que diferentes cenários de falhas produzam variados defeitos nessas aplicações.

3.1 Terminologia empregada na caracterização de falhas

Para a definição dos termos referentes à classificação de falhas, é tomado por base a taxonomia empregada por Avizienis et al. (2004) e repete-se a definição dos essenciais,

a seguir, para facilitar ao leitor. Está sendo empregado aqui o termo falha em correspondência ao que figura como *fault* (no inglês), naquela referência. Falhas **operacionais** aplicam-se às anomalias quanto à fase de ocorrência: o termo é usado para caracterizar aquelas que ocorrem durante o fornecimento do serviço, na fase de uso. Quanto à causa fenomenológica, as falhas podem ser **naturais**, quando são causadas por fenômenos naturais, sem a participação de pessoas, ou **humanas,** quando resultam de ações praticadas pelos seres humanos. Quanto ao objetivo, as falhas podem ser **maliciosas**, quando são introduzidas por humanos com o objetivo doloso, ou seja, de causar prejuízo ao sistema; ou **não-maliciosas**, as quais são introduzidas sem objetivo de prejudicar o sistema. As falhas ainda podem ser caracterizadas como **deliberadas** (**intencionais**), se resultam de uma decisão nociva; ou **não-deliberadas** (**não-intencionais**), quando são introduzidas de forma inconsciente ou inadvertida.

O escopo do presente artigo restringe-se ao tempo em que o sistema está em operação – de tal forma que todas as falhas podem ser enquadradas como operacionais. Não serão tratados aqui problemas que ocorrem durante a fase de projeto dos sistemas, que se contrapõem às falhas operacionais, na classificação quanto à fase de criação ou ocorrência.

Nodos que abandonam o sistema durante a transmissão (antes do seu encerramento) podem fazê-lo por perda de interesse ou por uma desconexão, por exemplo. No primeiro caso, pode-se falar em falha humana, não maliciosa e não deliberada: o usuário decide desvincular-se, mas, na maior parte das vezes, não percebe o seu papel no contexto da distribuição. No segundo caso, trata-se de falha natural. Em ambos casos, o comportamento pode ser representado através de colapso (*crash*). Cabe observar ainda que, nos casos de desconexão, é bastante provável que o usuário decida retornar ao sistema; dependendo de como o sistema tratar estes retornos, pode-se falar em colapso com recuperação (*crash-recover*). A corrupção de pacotes devido a ruídos no meio físico é outro exemplo de falha natural.

Falhas humanas maliciosas e deliberadas (ou intencionais) podem ser exemplificadas por nodos que retransmitem conteúdos inválidos (“lixo”) com o intuito de dissimular baixos níveis de contribuição; nodos que se associam a outros, privilegiando os “amigos” na redistribuição de pacotes; ou nodos que saem do sistema e retornam sob nova identidade para se desvincularem de má reputação obtida depois de terem exibido comportamentos inadequados, tais como os descritos na seção 4, subseções 4.2 e 4.3. Nesses casos, o comportamento é bizantino e complexo quanto à modelagem. *Softwares* mal configurados também correspondem a situações de comportamento bizantino, mas em boa parte dos casos são caracterizados como falhas humanas, não-maliciosas e não-deliberadas, pelo menos nos casos em que as deficiências de configuração devem-se à falta de conhecimento do usuário ao instalar o *software*.

3.2 Taxonomia comportamental dos usuários

No contexto de *live streaming*, as situações de falhas têm sido associadas majoritariamente ao comportamento dos usuários; é atribuída pouca atenção a anomalias nos programas de aplicações que lhes dão suporte. Este enfoque é mantido ao longo deste artigo. Um olhar possível sobre os participantes da rede de difusão de conteúdos pode subdividí-los quanto ao seu comportamento nos grupos citados e explicados a seguir.

Quanto à forma de participação – ou quanto ao comportamento cooperativo – os nodos podem ser caracterizados como corretos (incluem os altruístas), oportunistas (incluem os *freeloaders*, racionais, ou egoístas, que são todas denominações que exploram relaxamento nos mecanismos de controle do sistema) ou bizantinos.

Nodos **corretos** são aqueles que cumprem fielmente o protocolo definido para sua atividade – basicamente, eles solicitam dados na medida das necessidades e enviam dados de acordo com as requisições que recebem. Os nodos **altruístas** podem ser vistos como um subgrupo dos nodos corretos, os quais, nesse caso, estão dispostos a fornecer mais dados do que lhes é requisitado (contribuições espontâneas). Aiyer et al. (2005) sugerem que os nodos altruístas refletiriam apenas a existência de "bons samaritanos" e nodos fonte (*source*) em sistemas reais, mas não os permitem contribuir acima do especificado no protocolo.

O termo **oportunista** foi usado por Haridasan, Jansch-Pôrto e van Renesse (2008) para identificar nodos que tentam fornecer menos dados que eles forneceriam se exibissem um comportamento de nodos corretos, com a intenção de maximizar a relação entre a quantidade de dados obtida e o custo unitário destes. Essa relação pode ser maximizada quando eles buscam acesso aos dados que são distribuídos na rede, mas ignoram o atendimento às requisições recebidas. Não havendo mecanismos de controle, eles podem permanecer apenas desfrutando dos benefícios, enquanto o sistema sobreviver, já que o crescimento de usuários deste tipo condena o sistema à inanição. Outras caracterizações encontradas na literatura e que podem ser enquadrados no contexto de "oportunistas" aparecem sob as denominações: *freeloaders*, racionais, ou egoístas (*selfish*). Talvez seja possível agrupar o uso dos termos *freeloader* e *selfish*, citados na literatura, como sendo associados a um uso totalmente irresponsável dos recursos com o consumo de dados sem qualquer retribuição, se possível. O termo **racional** (tal como usado em BAR Gossip [Li, et al. 2006]) têm sido associado a nodos que seguem estritamente as regras de uso do sistema, se o seu desrespeito significar punição ou algum tipo de medida que implique em prejuízo na sua atividade de trocas. Logo, eles se desviam das regras apenas se com isso puderem maximizar seus ganhos.

Os nodos racionais têm por objetivo maximizar os seus benefícios de acordo com uma função-utilidade conhecida. Em economia, uma função-utilidade mede o nível de satisfação auferida por um consumidor a partir do consumo de um dado bem (ou um conjunto de bens). Em difusão de conteúdos, a função-utilidade está relacionada a custos que o nodo enfrenta com a sua participação; pode ser avaliada em ciclos de computação, área de armazenamento, largura de banda empregada, *overhead* associado ao envio e recebimento de mensagens, consumo de potência, sanções financeiras, ou vantagens decorrentes de sua participação no sistema, tais como armazenamento remoto, serviços de rede ou ciclos computacionais. Desvios com relação ao protocolo só ocorrem se isso acarretar benefício ao nodo racional, isto é, se o descumprimento fizer com que ele aumente a sua função-utilidade. Os nodos racionais também podem ser enquadrados como um subgrupo dos bizantinos pois, ao se desviar do protocolo definido, sua forma de atuação não faz parte do modelo do sistema.

Nodos **bizantinos** são aqueles que se comportam de uma forma completamente arbitrária, inclusive atuando em seu próprio prejuízo; logo o seu modelo de atuação é ignorado *a priori*. Possuem funções de utilidade arbitrárias e desconhecidas, decorrentes de configuração inadequada, mau funcionamento, problemas de programação, ou como

resultado de comportamento malicioso. Observe-se que o comportamento bizantino não pressupõe benefícios obtidos a partir do descumprimento do protocolo: é possível inclusive que ele siga uma estratégia que resulte em perdas incluindo ataques de negação de serviço (*Denial-of-Service*, DoS).

Em BAR Gossip [Li, et al. 2006], o primeiro artigo a propor um modelo que reúne nodos altruístas, racionais e bizantinos, os altruístas foram restringidos ao "cumprimento rígido do protocolo", sem poder contribuir de forma extra. Lá, eles diferenciam-se dos racionais apenas porque não se desviam do protocolo, nem com objetivo de ampliar benefícios do conjunto de participantes. Contribuições espontâneas iriam perturbar completamente o protocolo proposto. A convivência de nodos altruístas com oportunistas iria fazer com que os oportunistas explorassem, a seu favor, a "boa vontade" dos altruístas. Assim, com a existência destes, os nodos racionais não seriam obrigados a fazer trocas justas, tendo boas razões para desviar-se do seu comportamento definido pelo protocolo do sistema, já que a "exploração" dos altruístas aumentaria a função-utilidade dos racionais na obtenção de informações. Logo, protocolos que prevejam a convivência de nodos altruístas com racionais precisam ter características de adaptação dos níveis de contribuição dos nodos em função dos perfis de composição de participantes. Como é difícil conhecer esta composição *a priori*, apenas o monitoramento das condições da rede – ou da qualidade das informações recebidas – pode ser usado para esta adaptação, através de mecanismos de auditoria, por exemplo.

Quanto ao estado da conexão, pode-se falar em nodos ativos ou inativos. Os primeiros estão fisicamente conectados, participam das atividades recebendo pacotes e redistribuindo (de acordo com a sua forma de participação), e respondem a mensagens de controle. Os que se apresentam inativos, não respondem a qualquer tentativa de comunicação. Portanto, não consomem recursos mas tampouco contribuem na difusão de conteúdos. Precisam ser identificados e removidos já que distorcem os parâmetros relacionados à composição do *overlay*.

Quanto ao momento de conexão, os nodos podem solicitar participação no *overlay* a partir do início da transmissão, fazê-lo de forma tardia (durante a transmissão, após o seu início), podem retirar-se de forma prematura (quando a transmissão ainda está em andamento), ou ao final da transmissão. Os nodos que vêm e vão, respectivamente, no início da transmissão e ao seu final, não apresentam problemas ao sistema já que (se cumprirem o protocolo de difusão) participam de todas as fases, exceto quando um grande número deles chega simultaneamente (***flash crowd***). Os desafios estão relacionados aos procedimentos de manutenção do *overlay* que são necessários para incluir e atualizar os nodos que chegam tardiamente ao sistema, e para manter em nível aceitável o índice de continuidade nos nodos vizinhos ou dependentes dos que saíram prematuramente do sistema, principalmente quando isso ocorre em grupos (*churn*).

4. Falhas no contexto de *live streaming* (conceitos e técnicas usadas)

As instabilidades em conexões efetuadas através da Internet, o desinteresse dos usuários nas aplicações, e erros dos aplicativos ou no *software* podem determinar a desconexão dos usuários, resultando em comportamentos semelhantes ao que tradicionalmente é classificado como colapso ou falha de conexão (*link*). Por outro lado, o espírito "aproveitador" de usuários, que esperam apenas benefícios ao baixar pacotes e não distribuí-los em reciprocidade também vai causar problemas ao princípio cooperativo

proposto. Adicionalmente, participantes que tentam se aproveitar do interesse dos usuários para modificar pacotes e difundem, por exemplo, propaganda ou simplesmente lixo, também acarretam problemas às aplicações.

Assim, quando examinados quanto à forma de participação (ou espírito cooperativo), encontrar-se-á a possibilidade de que os nodos exibam comportamentos corretos (incluindo altruístas), oportunistas ou bizantinos. Mas não é possível identificar inequivocamente estes comportamentos em associação com os nodos, ou seja, "rotulá-los". Então avaliar a capacidade funcional do sistema diantes de diversas relações entre estes grupos, com estimativas quanto à sua composição percentual, é um enfoque possível. Os desvios de comportamento podem se dar por falhas humanas maliciosas ou não – mas a existência do interesse em prejudicar o sistema não é explícita; logo não pode ser alvo de análise. Assim, emprega-se nesse artigo apenas a divisão em falhas deliberadas (ou intencionais) ou não-deliberadas (não-intencionais).

Uma outra abordagem possível é quanto ao momento de conexão. Neste caso, embora as desconexões possam ocorrer voluntariamente, é difícil vincular este motivo ao modelo. Pode-se inferir motivação para a saída quando a qualidade é baixa, mas não há como confirmar as suspeitas. Após a conexão, e até que haja a desvinculação, os nodos são associados ao estado de conexão ativo.

4.1. Caracterização dos perfis básicos de falhas não intencionais

Os tipos de falhas não intencionais que se manifestam no contexto em estudo, em sua maioria, são decorrentes de: ruídos e congestionamentos no meio de comunicação, que causam erros no conteúdo dos pacotes recebidos ou perda (não recebimento) desses; ou fraca vinculação dos usuários, que faz com que haja uma dinâmica bastante intensa na composição do *overlay*. As entradas e saídas extemporâneas são tratadas como eventos indesejados e randômicos – portanto, enquadrados como falhas também. Esses aspectos são abordados a seguir.

Devido à natureza dos conteúdos multimídia, sua transmissão é bastante suscetível a erros [Meddour, Mushtag e Ahmed 2006]. Por outro lado, o grande volume de dados e a tolerância do usuário a um baixo nível de distorção fazem com que não se justifique incorporar codificação extensa a essa informação. Além disso, para que a recepção seja adequada, é necessário empregar um esquema bem planejado de codificação de vídeo confiável, de tal forma que a transmissão seja suficientemente flexível para adequar-se à dinâmica das redes P2P e sua heterogeneidade [Yang, et al. 2006]. Yang et al. dividem os conteúdos em dois níveis hierárquicos: enquanto as informações fluem adequadamente, ambos níveis são transmitidos. À medida que são identificados congestionamentos e possíveis atrasos decorrentes desses, apenas as informações de maior hierarquia são enviadas, reduzindo a qualidade do vídeo conforme percebido pelo usuário final. Os pacotes transmitidos precisam ainda incorporar codificação suficiente que permita a percepção da perda de pacotes ou sua alteração, a fim de que o protocolo possa tratar erros ou solicitar a retransmissão.

O comportamento de usuários finais da Internet é completamente imprevisível. Como participantes interessados em uma transmissão em curso, pode-se imaginar que a maior parte associa-se à difusão em seu horário inicial e desvincula-se ao seu final. Entretanto, não há observação suficiente sobre esse comportamento – e alguns estudos publicados não confirmam plenamente essa intuição. Adicionalmente, tomando-se por

base aplicações que envolvam programação de televisão, por exemplo, um bom percentual de usuários apenas se conecta para verificar se há algo de interesse ou se o programa alvo é “bom”, o que é julgado a partir de uma amostra. O terceiro aspecto é uma suspeita de que a desvinculação prematura de usuários pode ocorrer devido à insatisfação com a qualidade das informações recebidas, no que se refere a índice de continuidade, e atrasos no *playback*. Esses usuários podem tentar retornar novamente ao sistema e, dependendo dos critérios de composição do *overlay*, eles observam alguma melhora (ou não, piorando a qualidade de recepção), ou podem simplesmente desistir, abandonando definitivamente a rede. Mais tarde (seção 4.3), será visto que ainda existem outras razões para saída prematura causadas por falhas bizantinas.

Devido a essa natureza dinâmica dos usuários de redes P2P e à liberdade dos usuários quanto aos momentos de ingresso e abandono da rede (e, por conseqüência, da transmissão e redistribuição), sem notificação prévia aos demais, a gerência desse tipo de sistemas necessita de características que os façam reagir adequadamente às demandas provocadas por esse comportamento. As descontinuidades locais na distribuição dos pacotes não devem afetar a percepção daquilo que é recebido pelos demais usuários, ou seja, a regularidade no fluxo de pacotes deve ser mantida em nível suficiente para atenuar os impactos sobre a taxa de *playback* durante a sessão correspondente. Há necessidade de empregar mecanismos que sejam simultaneamente robustos e adaptativos para lidarem com essa dinâmica da rede. Alguns trabalhos mencionam esses cuidados, mas talvez um dos maiores problemas seja o de que ainda não se dispõe de uma modelagem efetiva associada a este comportamento que possa ser usada para comprovar a eficiência das técnicas propostas.

Os dados referentes ao comportamento dos usuários ainda são bastante limitados, já que a maior parte das aplicações de *live streaming* ainda opera em níveis experimentais. Assim, a obtenção de dados ainda pode estar contaminada por tendências, motivação da população para associar-se à transmissão, entre outras.

Liao et al. (2006) apresentam alguns dados relacionados a experimentos conduzidos com o AnySee, um sistema desenvolvido para distribuição de pacotes em *live streaming* que busca ter como características básicas eficiência e escalabilidade. Este sistema vem sendo usado na China e, de acordo com a referência citada, mais de 60 mil usuários têm tido acesso a programas de televisão, filmes e conferências acadêmicas. Segundo eles, não há uma relação comprovada entre o interesse dos usuários (e conseqüente permanência no sistema) e atrasos na distribuição de informações. Entretanto, estes dados podem ser questionados diante dos aspectos explicados a seguir: a) as amostragens são pequenas e talvez com um certo componente de tendência: foram realizadas em sete momentos diferentes (*hot periods*) sobre três programas selecionados. Não há informações sobre o conjunto de usuários observados e em que condições foram colhidas as amostras, por exemplo. Mas há menção a 7200 usuários de mais de 40 universidades em 14 cidades da China - ou seja, não se tratavam de usuários domésticos comuns; b) os atrasos considerados para tirar conclusões sobre o comportamento dos usuários estão na ordem de dezena de segundos. São valores médios, sem qualquer informação adicional sobre o desvio padrão e intervalo de confiança - portanto, são dados estatisticamente muito pobres. Assim, os cerca de 10% de usuários que abandonam o sistema não causam um impacto significativo sobre as demais características do sistema e ajudam a compor os gráficos apresentados como resultados experimentais.

Liu, Rao e Zhang (2008) apresentam um ponto de vista diferente de Liao et al. (2006). Eles afirmam que, se a qualidade do vídeo não é adequada durante ao período inicial de distribuição, é bastante provável que o usuário abandone o sistema, sendo possível inclusive a ocorrência de *churn*. Esse abandono maciço tende ainda a realimentar os problemas de distribuição e incentivar mais usuários a se desvincularem. Além deste comentário, eles ainda fazem algumas suposições estatísticas referentes ao ingresso de usuários no início de uma transmissão: se um milhão de interessados se reunirem ao sistema no início da transmissão – nos primeiros 100 segundos – a taxa de pico de chegada será de 10.000 nodos por segundo. Trata-se de um problema real de escalabilidade para associar eficientemente à rede os nodos interessados.

4.2. Um ambiente para falhas intencionais

Matteo Dell'Amico (2006) comenta alguns problemas decorrentes das características das redes P2P. Um deles é a ausência de conhecimento global sobre a rede: em ambientes grandes, além de ser provavelmente desprovido de interesse, é impossível aos participantes manterem conhecimento sobre cada interação que ocorre no sistema. Enquanto isso simplifica a operação, abre um campo expressivo para problemas de segurança. Problemas apontados por ele são ataques do tipo *whitewashing* e conluio (*collusion*), que serão vistos nesta seção (subseção 4.3).

Em ambientes distribuídos abertos, tal como ocorre com os sistemas construídos sobre a Internet, e principalmente naqueles que cujos participantes são conjuntos de nodos altamente dinâmicos, abre-se a possibilidade a uma entidade qualquer aparecer sob diferentes identidades. Isso se constitui em ameaça à segurança e precisa se tratada por mecanismos próprios. É preciso lembrar que boa parte das hipóteses empregadas tradicionalmente em dependabilidade repousam sobre a premissa de que os participantes são independentes e detêm identidades diversas. Entretanto, será visto que alguns cenários resultantes de ataques não se enquadram nesse tipo de hipótese [Androutsellis-Theotokis e Spinellis 2004].

Adicionalmente, pode-se dizer que as falhas intencionais constituem a classe mais rica em função das possibilidades de comportamento. Não se pode antecipar todas as variantes que a imaginação humana irá produzir. Nessa seção, serão caracterizados as formas de comportamento mais conhecidas, tipos de impacto que elas podem causar e algumas abordagens que têm sido usadas para conter ou minimizar os prejuízos decorrentes destes tipos de ataques.

Enquanto seguem estritamente o protocolo, os nodos racionais (subseção 3.2) não deveriam oferecer problemas ao sistema, já que um protocolo bem construído assegura que as ações mantenham um balanço equilibrado entre contribuição e obtenção de conteúdos. Entretanto, eles seguem as regras apenas se correrem riscos no descumprimento e enquanto não descobrirem desvios admissíveis. Mecanismos para restringir as ações oportunistas incluem o uso de auditores, que controlam os níveis de contribuição dos participantes, e o enfoque *tit-for-tat*, no qual um nodo apenas envia dados a outro nodo se recebe um montante em retorno ou pagamento. Mas uma rede aberta e dinâmica (que oferece a possibilidade de retorno a participantes, o que inclui ex-infratores) dificulta o controle dos indivíduos e o emprego de mecanismos disciplinadores. Por consequência, não estimula a manutenção do papel correto, sem desvios. Até o momento, não se tem conhecimento de uma solução adequada substitutiva ao *tit-for-tat* porque, em atividades de baixíssimo valor financeiro por

unidade de transferência (remuneradas através de *micropayments*) como é o caso dos pacotes em *live streaming*, a manutenção dos mecanismos é muito cara frente à aplicação. Técnicas de auditoria ativadas apenas quando a qualidade dos índices de continuidade estão descendo a patamares perigosos (próximos dos níveis mínimos aceitáveis) foram sugeridos por Haridasan, Jansch-Pôrto e van Renesse (2008). Os resultados apresentados são promissores e a estimativa de impacto no custo do sistema mostra níveis adequados. Ainda é importante ressaltar que qualquer desvio no comportamento de um nodo racional o desloca para um comportamento bizantino – sendo que assim a antecipação do modelo de comportamento fica bastante complexa.

4.3. Caracterização de perfis básicos de falhas intencionais (bizantinos)

Uma das razões para usuários terem interesse em exibir identidades múltiplas é livrar-se de ações inadequadas cometidas no passado. Essas ações podem ter sido, por exemplo, a omissão em contribuir na distribuição de conteúdos (referidos como *omission attacks*, [Haridasan e Van Renesse 2006]) e serem flagrados pelos mecanismos de incentivos empregados no sistema. Ou pode ser que ações indevidas e o descumprimento de promessas, medidos por mecanismos de avaliação, os classifique mal através de parâmetros de reputação utilizados. O retorno ao sistema sob nova identidade poderia ser uma alternativa de voltar a desfrutar dos benefícios; outra é o uso simultâneo de múltiplas identidades que sirvam para atuação sob perfis diversos.

Quando há liberdade na criação de identidades, e essas podem ser usadas de forma proveitosa por quem as detêm, alguns usuários podem valer-se dessa fragilidade do sistema em benefício próprio criando múltiplas identidades. Esse tipo de ação foi definida por Douceur (2002), sob a denominação de ***Sybil attack***. Hipoteticamente, se uma única entidade defeituosa pode atuar apresentando-se através de diversas identidades, ela pode controlar uma parte significativa do sistema, minando os meios de redundância usados para sua proteção. Uma forma de prevenir esses ataques é manter identidades certificadas através de uma agência confiável, o que só é obtido em cenários reais através de uma autoridade lógica centralizada.

Infraestruturas centralizadas ou de gerência de identidade podem, portanto, minorar o problema; entretanto são soluções difíceis de implementar na prática [Piatek, Anderson e Krishnamurthy 2007]. Por consequência, esses autores propõem que à simples obtenção de identidades não sejam atribuídos quaisquer dados ou elementos de valor (como forma de *bootstrapping*), de tal forma que o usuário não obtenha vantagem dessa ação. Eles propõem o uso de mecanismos de incentivo, mas resta ainda em aberto a forma de inicialização de novos participantes. Num contexto de *live streaming*, o usuário pode obter vantagem de múltiplas identidades se, ao entrar no sistema, ele receber imediatamente conteúdos de atualização de forma que possa integrar-se às demais atividades. Por outro lado, sem dados, um nodo ingressante tampouco pode contribuir com os seus vizinhos.

Uma outra alternativa seria o uso de técnicas de identificação que consumam muitos recursos – de tal forma que não sejam atrativas para uso prático pelos atacantes [Androutsellis-Theotokis e Spinellis 2004]. Entretanto, há necessidade de considerar os custos envolvidos aos usuários regulares do sistema (usuários P2P comuns), e não menosprezar a capacidade daqueles que desejam servir-se maliciosamente dele.

Para que seja mantida a operação correta do *overlay*, é necessário que os nodos corretos mantenham seu potencial de comunicação e repasse de mensagens através dos *links*. Esta capacidade é o alvo de participantes maliciosos que isolam nodos corretos, ou os "eclipsam" (***eclipse attacks***) através do preenchimento da tabela de vizinhos de um membro correto com endereços de participantes maliciosos. Assim, se o atacante controlar uma fração significativa de conjuntos de vizinhos de nodos corretos, ele impede este nodo de receber mensagens endereçadas a ele. A expansão exagerada deste controle poderia resultar inclusive no controle integral do tráfego no *overlay* pelo atacante [Singh, et al. 2004]. Este ataque pode ser iniciado a partir de um Sybil *attack*, seguido do preenchimento da tabela de vizinhos de nodos corretos com identificação de nodos aparentemente distintos. Se combinado com o protocolo de manutenção para a substituição de nodos defeituosos, pouco a pouco os maliciosos vão dominando a rede.

Defesas contra este tipo de ataques incluem a proposta de Castro et al. (2002), voltada para *overlays* estruturados, que impõem o uso de restrições estruturais fortes em sua composição. Uma desvantagem é que estas restrições impedem a implantação de otimizações visando melhorar o desempenho. Singh et al. (2004) propuseram uma outra forma para evitar este tipo de ataque pelo estabelecimento de limites (*thresholds*) conjuntos na análise dos graus de entrada e saída dos membros da comunidade P2P. Juntamente com a implementação deste mecanismo, é necessário um sistema de auditoria que impeça os nodos maliciosos de mentirem sobre os valores dos seus graus de entrada e saída. De acordo com os autores, a técnica não é particular a qualquer tipo de estrutura, além de permitir o uso de otimizações baseadas em métricas tais como proximidade física, necessária para a eficiência do *overlay*.

Um outro tipo de ataque – ***whitewashing*** – consiste da obtenção de nova identidade na vinculação (ou retorno) à rede, simulando ser um neófito ao desfazer-se do histórico anterior. Esse procedimento faz com que nodos possam desvincular-se de má reputação obtida a partir de "mau comportamento". Isso tem sido uma operação cujo custo (tipicamente) é baixo, o que acaba sendo do interesse de nodos maliciosos que apresentam histórico de não-contribuição ou de ações indevidas. É desincentivado, entretanto, quando nodos recentes não recebem privilégios de ingresso, mas precisam prestar serviços ou pagamentos para receber benefícios [Friedman e Resnick 2001]. Procedimentos simplificados e rápidos na saída e entrada de nodos funcionam como convite a esse tipo de procedimento. Portanto, buscar uma relação de compromisso na qual as facilidades não estimulem atacantes mas apresentem exigências adequadas a usuários regulares (honestos) é um desafio.

Na classe de ataques por falsificação (*forgery*), citado por Haridasan e van Renesse (2006) estão os ataques que envolvem a fabricação e adulteração de dados que estão sendo difundidos através do sistema. Os mecanismos de contenção deste tipo de ataques operam com base em uma infraestrutura de chave pública. O problema é o custo deste mecanismo de assinaturas, muitas vezes proibitivo numa aplicação de *streaming*, sendo reduzidos a protocolos mais simples para autenticação de dados.

Ataques por **conluio** (***collusion attacks***) ocorrem como resultado da conspiração de um grupo de participantes maliciosos; eles realizam, de forma combinada, uma ação, ou conjunto dessas, de forma a obterem benefícios para si ou para participantes do grupo. Essas ações podem ser de diferentes naturezas: por exemplo, os nodos podem introduzir informações incorretas em seu histórico, de forma a criar ataques que visam

aumentar ou diminuir artificialmente a reputação de alguns nodos. Um reforço para a resiliência com relação a este tipo de ataques é obtido pela associação de maior peso aos votos gerados por nodos de maior reputação no sistema [Dell'Amico 2006]. Na difusão de arquivos ou de *streaming*, a atuação de um grupo em conluio tipicamente se dá pelo repasse de conteúdos ou informações apenas a participantes que fazem parte do grupo, deixando à míngua os demais.

Os ataques por conluio não precisam ser necessariamente efetivados por participantes diversos, mas podem resultar da criação de múltiplas identidades espúrias relacionadas a um participante – e neste caso enquadram-se no conceito de *Sybil Attack*.

5. Comentários em aspectos de modelagem de falhas

Estabelecer modelos correspondentes adequados para a representação dos diferentes tipos de falhas não é tarefa fácil, devido à imprevisibilidade de comportamento, do tipo de mecanismo que será alvo de ataque e do próprio objetivo dos atacantes (por exemplo, nodos bizantinos podem "decidir" atuar em seu próprio prejuízo). Por esse motivo, autores têm estimado algumas situações particulares a fim de estudarem o impacto de falhas sobre os sistemas de *live streaming*.

Uma outra dificuldade é estimar *a priori* a distribuição dos perfis de desvio em diversos sistemas – por exemplo, estabelecendo a composição entre nodos corretos, racionais e bizantinos numa rede aberta. De acordo com os autores de BAR Gossip [Li, et al. 2006], os participantes racionais corresponderiam à maioria dos participantes em um sistema P2P através de múltiplos domínios administrativos. Entretanto, esta afirmativa não é sustentada por qualquer tipo de avaliação. Pode-se imaginar que seja resultado do protocolo estabelecido inicialmente, e que a maioria dos usuários instala em sua configuração básica (*default*); mesmo assim, não há como antecipar desvios sobre este quadro.

Assim, na tentativa de estudar sistemas cujos perfis de composição sejam variados, alguns trabalhos mais recentes modelam as interações entre os diferentes perfis de usuários através da teoria conhecida como "Equilíbrio de Nash" [Nash 1950] e/ou de modelos originados na Teoria de Jogos. Uma das desvantagens das propostas baseadas no trabalho de Nash é que os seus fundamentos repousam sobre um ambiente limitado (ou fechado) e conjuntos de regras que são conhecidos por todos os participantes. Assim, conluios (nos quais existe coordenação de nodos através de regras "extra-sistema" e válidas apenas para sub-grupos) não se enquadram nesse modelo. A teoria de jogos parece ser promissora já que vem sendo ampliada com inspiração em atividades conduzidas através da Internet, e portanto num cenário bastante semelhante aos de difusão de conteúdos, sejam eles ao vivo ou não. Mas a modelagem de aplicações de rede e de difusão de arquivos, através desta teoria, ainda é incipiente.

Outras alternativas mais simplificadas têm sido encontradas em trabalhos que buscam estudar o efeito do comportamento malicioso em ambientes de difusão de *live streaming*. Num exemplo usado para simulações de comportamentos maliciosos, Haridasan e van Renesse (2006) consideram quatro tipos de ataques: a) os nodos agem como totalmente falhos, não requisitando pacotes nem atendendo solicitações; ou b) solicitam pacotes mas não os repassam; ou c) eles sobrecarregam os vizinhos com a solicitação de grande quantidade de pacotes (tantos quantos forem possíveis); ou ainda d) eles sobrecarregam os vizinhos com a solicitação de grande quantidade de pacotes e

não os repassam. Portanto, a última opção é a que mais prejudica o sistema. Cada um dos cenários usado separadamente é estudado com a variação percentual dos participantes maliciosos.

Em BAR Gossip, Li, et al. (2006) modelaram nodos maliciosos para estudar um cenário de conluio perfeito, no qual todos nodos participantes do ataque difundem as atualizações, de forma imediata ao recebimento, apenas ao seu grupo. Isso coloca em risco a distribuição de informações aos nodos não participantes do grupo, pois não existe mais a reciprocidade nas trocas. No cenário lá apresentado, os nodos participantes do conluio são considerados racionais, enquanto os demais são altruístas. Isso significa que os conspiradores apenas recebem pacotes (enquanto for do seu interesse, seguindo o protocolo racional) quando interagem com nodos não-conspiradores, enquanto que os externos ao grupo seguem fielmente o protocolo. Os resultados ali apresentados mostram que o sistema torna-se inutilizável apenas quando os nodos conspiradores atingem 50% da população. Ao negar trocas com os não envolvidos na trama, os nodos em conluio vão minando a distribuição de dados: num caso limite, eles terão eliminado todos os nodos não envolvidos e terão que atuar em trocas de forma completa para continuar a sobreviver.

Os próprios autores [Li, et al. 2006] constataram problemas associados a grupos muito grandes, os quais necessitam de maior largura de banda e geram maior latência na difusão das informações a todos os membros do grupo. Além disso, uma análise da modelagem proposta àquele sistema permite inferir que ela favorece o cenário de conluio na medida em que os nodos externos ao grupo continuam repassando pacotes apesar de não obterem qualquer vantagem nas trocas ou mesmo usufruírem de sua realização (uma vez que recebem apenas lixo). Um cenário com nodos apenas racionais (com um subgrupo envolvido no conluio) não permitiria tal expansão, pois o grupo rapidamente perderia seus “alimentadores”. Um aspecto estranho na análise apresentada é que os participantes do conluio são classificados como racionais; entretanto, na medida em que difundem dados aos participantes do grupo de forma “gratuita”, eles não estão maximizando o seu benefício individual, o que é de certa forma contraditório no modelo (racional) considerado. Um nodo participante do conluio “esforça-se” para obter dados dos externos ao grupo mas atua como “um bom samaritano” frente ao seu grupo, doando dados.

Ao modelar nodos bizantinos para avaliar seu impacto sobre os sistemas, Li et al. (2006) pressupõem que o objetivo desses é inverso ao de qualquer participante racional, fazendo com que eles aumentem o custo e diminuam seus benefícios com relação ao perfil racional. Certamente, trata-se de um modelo bastante simplificado frente às possibilidades que um nodo bizantino pode “criar”, mas é um ponto de partida.

6. Conclusões

Quando comparados aos sistemas tradicionais de difusão de arquivos via Internet, as aplicações de *live streaming* agregam exigências por suas condições de tempo real e escalabilidade: os pacotes enviados precisam chegar ao grande número de usuários com regularidade e pequenos atrasos. Assim as soluções de gerência do sistemas e os mecanismos usados para conter os efeitos das falhas precisam ser eficientes e leves, de forma a não prejudicar o sistema.

Foram discutidos os principais tipos de falhas que afetam as aplicações de *live streaming*, incluindo um enfoque comportamental e outro temporal, além de caracterizar as principais modalidades de ataques que são usadas de forma intencional. Também foram enfocados os problemas de modelagem de falhas nestes ambientes, quanto às abordagens e falta de dados reais (ainda) neste escopo de aplicações.

Há vários problemas que restam em aberto no tratamento de falhas deste grupo de aplicações e tratam-se ainda de desafios bastante interessantes, embora novas propostas estejam surgindo rapidamente numa área de pesquisa que se apresenta altamente dinâmica.

Referências

Androutsellis-Theotokis, S. and Spinellis D. (2006) "A Survey of *Peer-to-peer* Content Distribution Technologies." *ACM Computing Surveys* v. 36, n. 4, December, pp. 335-371.

Avizienis, A., Laprie, J.-C., Randell, B. and Landwehr, C. (2004) "Basic Concepts and Taxonomy of Dependable and Secure Computing." *IEEE Transactions on Dependable and Secure Computing*, v. 1, n. 1, pp. 11-33.

Awerbuch, B., Holmer, D., Nita-Rotaru C., and H. Rubens (2002) "An On-demand Secure Routing Protocol Resilient to Byzantine Failures". *Proceedings of the 1st ACM Workshop on Wireless Security*. Atlanta, GA. pp. 21-30.

Ayer, A. S., Alvisi, L., Clement, A., Dahlin, M. Martin, J. P. and Porth, C. (2005) "BAR Fault Tolerance for Cooperative Services." *SIGOPS Oper. Syst. Rev.*, v.39, n. 5, October, pp. 45-58.

Castro, M., Druschel, P., Ganesh, A., Rowstron, A. and Wallach, D. S. (2002) "Secure Routing for Structured P*eer-to-Peer* O*verlay* Networks." *Proc. of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*. Boston, MA. pp. 299-314.

DaSilva, L. A. and Srivastava, V. (2004) "Node Participation in Ad Hoc and Peer-to-Peer Networks: A Game-Theoretic Formulation". *Proceedings of the First Workshop on Games and Emergent Behaviors in Distributed Computing Environments*. Birmingham, UK.

Dell'Amico, M. (2006) "Neighbourhood Maps: Decentralised Ranking in Small-World P2P Networks." *Proceedings of the Hot Topics in Peer-to-peer Systems (HoTP2P2006)*.

Douceur, R. (2002) "The Sybil *attack*." *Proceedings of the First International Workshop on Peer-to-peer Systems (IPTPS)*. Springer Berlin / Heidelberg, Cambridge, MA, USA, pp. 251-260.

Fodor, V., and Dán, G. (2007). "Resilience in Live *Peer-to-peer* Streaming." *IEEE Communications Magazine*, v. 45, n. 6, June, pp. 116-123.

Friedman, E. J., and Resnick, P. (2001) "The Social Cost of Cheap Pseudonyms." *Journal of Economics & Management Strategy* v.10, n.2, August, pp. 173-199.

Ge, Z., Figueiredo, D. R., Jaiswal, S., Kurose, J., Towsley, D. (2003) "Modeling Peer-Peer File Sharing Systems". *Proceedings of the Twenty-Second Annual Joint*

Conference of the IEEE Computer and Communications Societies (INFOCOM 2003), vol 3, pp. 2188-2198.

Hales, D. (2004) "From Selfish Nodes to Cooperative Networks - Emergent Link-Based Incentives in Peer-to-Peer Networks". *Proceedings of the Fourth International Conference on Peer-to-Peer Computing*, pp. 151-158.

Haridasan, M., Jansch-Pôrto, I., and van Renesse, R. (2008) "Enforcing Fairness in a Live-Streaming System." *Proceedings of SPIE: Multimedia Computing and Networking*. v. 6818. San Jose, CA.

Haridasan, M., and van Renesse, R. (2006) "Defense Against Intrusion in a *Live streaming* Multicast System." *Proceedings of the 6th IEEE International Conference on Peer-to-peer Computing (P2P)*. Cambridge, UK, pp. 185-192.

Li, H. C., et al. (2006) "BAR Gossip." *Proceedings of the 7th Symposium on Operating System Design and Implementation (OSDI '06)*. Seattle, WA, pp. 191-204.

Liao, X., Jin, H., Liu, Y., Ni, L. M., and Deng, D. (2006) "Anysee: *Peer-to-Peer Live Streaming*." *25th IEEE Intl. Conf. on Computer Commun (INFOCOM)* pp. 1-10.

Liu, J., Rao, S. G. and Zhang, H. (2008) "Opportunities and Challenges of *Peer-to-peer* Internet Video Broadcast." *Proceedings of the IEEE*, v. 96, n. 1, Jan., pp. 11-24.

Magharei, N., and Rejaie, R. (2007) "PRIME: *Peer-to-peer* Receiver-drIven *Mesh*-based Streaming." *Proceedings of the 26th Conference on Computer Communications (INFOCOM)*. Anchorage, Alaska, pp. 1415-1423

Meddour, D.-E., Mushtag, M. and Ahmed, T. (2006) "Open Issues in P2P Multimedia Streaming." *Proc. IEEE ICC: Multimedia Commun. Workshop (MultiCom)*.

Nash, John (1950) "Equilibrium Points in n-person Games." *Proceedings of the National Academy of Sciences* v. 36, n. 1, pp. 48-49.

Pai, V., Kumar, K., Kamilmani, K., Sambamurthy, V. and Mohr, E. (2005) "Chainsaw: Eliminating Trees from *Overlay* Multicast." *Proceedings of the 4th International Workshop on Peer-to-peer Systems (IPTPS)*. Ithaca, NY, pp. 127-140.

Piatek, M., Anderson, T. and Krishnamurthy, A. (2007) "A Case for Holistic Incentive Design." *Proc. Workshop Future Directions in Distr. Computing (FuDiCo III)*.

Shneidman, J., Parkes, D. C. (2003) "Rationality and Self-Interest in Peer to Peer Networks". *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*. Berkeley, CA, USA, pp. 139-148

Singh, A., Castro, M., Rowstron, A. and Druschel P. (2004) "Defending against Eclipse *attacks* on *overlay* networks." *Proceedings of the 11th ACM SIGOPS European Workshop*. Leuven, Belgium.

Yang, G.-H., Shen, D., Yang, D. and Li, V. O. K. (2006) "Adaptive Video Streaming over Multi-channel Ad Hoc Networks." *Global Telecommunications Conference IEEE (GLOBECOM'*06). pp. 1-5.

Zhang, X., Liu, J., Li, B. and Yum, T.-S P. (2005) "CoolStreaming/DONet: A Data-Driven *Overlay* Network for Efficient Live Media Streaming." *Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)* v. 3. Miami, FL, USA, pp. 2102-2111.

Geração Automática de Objetivos e Casos de Teste a partir de Redes de Petri Orientadas a Objetos*

André L. L. Figueiredo and Patrícia D. L. Machado and Emanuela G. Cartaxo and Jorge C. A. Figueiredo and Paulo E. S. Barbosa

[1]GMF/DSC - Universidade Federal de Campina Grande (UFCG), Brasil

{andrel, patricia, emanuela, abrantes, paulo}@dsc.ufcg.edu.br

Resumo. *Este artigo apresenta um método para a seleção e geração automática de objetivos de teste e seus respectivos casos de teste a partir de modelos abstratos de sistemas reativos. É dado enfoque a modelos de Redes de Petri Orientadas a Objetos (RPOO). Configurações e seqüências de ativação das redes são adotadas como objetivos de teste. Estes são convertidos para os Sistemas de Transição Rotulados (LTSs) correspondentes que podem então ser usados como entrada para ferramentas de geração de casos de teste.*

1. Introdução

Teste funcional é um tipo de teste de software que objetiva observar se um dado sistema contempla de forma correta as funcionalidades descritas para ele [Jorgensen 2002]. Com base na sua respectiva especificação, um sistema é exercitado e suas funcionalidades são observadas através de seu comportamento, onde os possíveis defeitos são revelados. Dentre os diversos tipos de teste, um tipo de teste funcional se destaca - teste formal. Este tipo de teste permite avaliar, de forma automática, se um determinado software está de acordo com sua especificação formal [Tretmans 1999, Gaudel 1995]. Neste sentido, é possível observar de forma automática se um software realiza corretamente as funcionalidades contidas em sua especificação.

Uma das técnicas de teste formal mais aplicadas para a geração dos casos de teste é a baseada em verificação de modelos [Clarke et al. 1999]. Esta técnica tem o nome de teste baseado em propriedades pois, assim como na verificação de modelos, as propriedades que se deseja verificar são especificadas formalmente e servem de entrada para a técnica. Por essa razão, esta técnica também recebe o nome de teste formal com propriedades explícitas, uma vez que as propriedades são elaboradas pelo testador. Se por um lado a verificação de modelos visa percorrer uma especificação em busca de falta de conformidade entre esta e uma determinada propriedade, a geração de casos de teste baseada em propriedades visa percorrer esta mesma especificação com o intuito de selecionar partes desta especificação que satisfazem a propriedade. Com isto, teste formal pode se utilizar de várias técnicas bem consolidadas de verificação de modelos, como por exemplo, as utilizadas para percorrer, comparar e reduzir grafos [Fernandez et al. 1996].

As propriedades utilizadas para a geração dos casos de teste são chamadas de objetivos de teste (*test purpose*). Estes constituem partes do sistema modelado as quais se deseja testar. Assim, dado o modelo do sistema e um objetivo de teste, a técnica gera casos de teste que visam testar, especificamente, as partes descritas pelo objetivo de teste.

*Este trabalho recebeu apoio financeiro do CNPq/FAPESQ, Projeto 060/03.

Pesquisas em seleção e geração de casos de teste baseada em objetivos de teste já produziram resultados importantes, incluindo modelos formais e ferramentas. Como exemplo, podemos citar a ferramenta TGV [Jard and Jéron 2004] desenvolvida pelo projeto AGEDIS [Hartman and Nagin 2004] e a ferramenta LTS-BT [Cartaxo et al. 2008]. No entanto, as soluções atuais demandam que o testador defina os objetivos, usualmente, de forma *ad-hoc* e escritos diretamente em uma notação de baixo nível tal como sistemas de transição rotulados (LTSs) que são amplamente adotados como modelos para a geração de casos de teste. Isto pode limitar a aplicabilidade de objetivos de teste, bem como comprometer a qualidade e confiabilidade de processos de teste. Objetivos de teste são essencialmente abstrações sobre comportamentos e propriedades dos modelos.

Este artigo apresenta um método para a seleção e geração automática de objetivos de teste em modelos LTSs a partir de modelos formais abstratos de sistemas reativos. É dado enfoque a modelos de Redes de Petri Orientadas a Objetos (RPOO)[Guerrero 2002]. Configurações e seqüências de ativação das redes, possivelmente selecionadas durante o processo de simulação, são adotadas como objetivos de teste, garantindo que comportamentos simulados poderão ser testados na implementação final. Estes são convertidos automaticamente para os LTSs correspondentes que podem então ser usados como entrada para ferramentas de geração de casos de teste. O método utiliza ferramentas para verificação e simulação de modelos a fim de validar e gerar os modelos do sistema e dos objetivos de teste como LTSs: o modelo gerado para verificação é o modelo a partir do qual casos de teste serão gerados. Um estudo de caso no domínio de aplicações de agentes móveis é apresentado a fim de ilustrar os passos de aplicação do método. Os objetivos de teste gerados são consistentes e completos com relação ao modelo do sistema. Finalmente, o conjunto de casos de teste é consistente.

O artigo está estruturado da seguinte forma. A Seção 2 apresenta fundamentação teórica sobre teste formal, objetivos de teste e RPOO, incluindo ferramentas. A Seção 3 apresenta o método proposto. A Seção 4 ilustra a aplicação do método através de um estudo de caso. Por fim, a Seção 5 discute trabalhos relacionados e a Seção 6 apresenta comentários finais sobre este trabalho e apontadores para trabalhos futuros.

2. Fundamentação Teórica

Esta seção apresenta conceitos fundamentais e ferramentas utilizadas neste trabalho com enfoque em teste formal e redes de Petri orientadas a objetos.

2.1. Teste Formal e Objetivos de Teste

Tretmans [Tretmans 1999] propôs um framework para o uso de métodos formais em teste de conformidade (teste formal). Este framework abstrai os conceitos presentes no processo de teste de conformidade, além de definir uma estrutura que nos permite tratar o processo de teste de uma maneira formal. Nesta seção, conceitos básicos em teste formal, com base neste framework serão brevemente descritos.

Um processo de teste formal possui as fases de geração e execução dos testes, ambas possivelmente automáticas. Durante a geração, a especificação do sistema a ser testado é analisada com o intuito de se obter informações sobre tal sistema que permite, ao testador, decidir sobre sua correção. Neste sentido, são gerados os seguintes artefatos: (i) casos de teste, descrevem um comportamento esperado do sistema; (ii) dados de teste,

as entradas do sistema para a execução dos casos de teste; (iii) oráculos, especificação de procedimentos capazes de verificar se uma dada execução do sistema (estimulada pelos dados de entrada) está de acordo com o comportamento definido pelos casos de teste. Na fase de execução, casos de teste gerados são implementados e executados sobre o sistema sendo testado. Ao final, os resultados da execução dos testes são analisados. O foco deste trabalho é a geração de casos de teste com base em objetivos de teste.

Um conceito bastante utilizado na comunidade de teste formal é a relação de conformidade chamada ioco [Tretmans 1999], que se baseia nas entradas e saídas (input/output) dos modelos a serem verificados. Esta relação define que um modelo de uma implementação está de acordo com sua especificação se, a partir de qualquer instante, para uma mesma seqüência de entradas, tanto o modelo quanto a especificação produzirão a mesma saída. Na verdade, não é necessário que as saídas sejam iguais, é preciso que o conjunto de saídas produzido pelo modelo esteja contido no conjunto produzido pela especificação, uma vez que poderemos estar tratando de sistemas não-determinísticos.

Para conjuntos de casos de teste denominados de *completos* os casos de teste são executados com sucesso se e somente se a implementação está em conformidade com a especificação. Contudo, estes conjuntos são muito difíceis de serem construídos na prática. Portanto, busca-se a construção de conjuntos de teste chamados *consistentes* os quais só rejeitam uma implementação se esta de fato não está em conformidade com a especificação. No entanto, implementações que não estão em conformidade também podem ser aceitas. Em outras palavras, podem detectar apenas não-conformidade.

Em teste formal, os sistemas são especificados utilizando-se uma determinada linguagem, muito possivelmente uma linguagem gráfica e já habitual no desenvolvimento de tais sistemas como, por exemplo, UML. Caso esta linguagem não seja baseada em algum formalismo, há a necessidade de uma formalização (pois estamos tratando de geração automática) destes modelos, caso contrário, geralmente é feita uma representação (também chamada de simulação ou transformação) em algum formalismo base. Esta representação ocorre devido ao fato de que a maioria das técnicas e ferramentas para a geração de casos de teste utilizam, como entrada, especificações em formatos bases, por exemplo grafos de transições rotuladas (*Labelled Transition System* - LTS). Os modelos utilizados pelas ferramentas de geração os casos de teste são, em geral, LTS's.

Os objetivos de teste são especificados, em geral, também neste formalismo base. No entanto, podem sofrer um processo de transformação semelhante à especificação do sistema, caso sejam descritos com algum outro tipo de linguagem. Com as duas especificações (sistema e objetivo de teste), o sintetizador de teste irá gerar uma especificação do sistema que contém apenas as partes consideradas pelo objetivo de teste, ou seja, irá selecionar na especificação do sistema apenas as partes descritas pelo objetivo de teste. Após isto, com esta especificação sintetizada, a ferramenta de geração de casos de teste se encarregará de gerar os diversos casos de teste, geralmente na mesma linguagem utilizada na especificação sintetizada, porém diagramas de seqüencia, *state charts* e outros também são utilizados.

TGV (*Test Generation with Verification technology*) [Jard and Jéron 2004] é uma ferramenta de teste funcional que gera automaticamente casos de teste de conformidade para sistemas reativos. Além de se basear em fundamentação teórica sólida, em particular

a teoria ioco apresentada em [Tretmans 1999], a ferramenta já foi aplicada com sucesso na indústria [Hartman and Nagin 2004]. A geração de casos de teste é baseada em técnicas de verificação de modelos tais como produto síncrono e verificação *on-the-fly*.

2.2. Redes de Petri Orientadas a Objetos

RPOO (Redes de Petri Orientadas a Objetos) [Guerrero 2002] é uma linguagem de modelagem e especificação de sistemas distribuídos e concorrentes baseada em redes de Petri e orientação a objetos. RPOO pode ser vista como uma extensão OO para redes de Petri de alto nível ou como um meio de prover semântica formal de concorrência a modelos OO. A idéia básica de RPOO é permitir a modelagem em dois níveis diferentes de abstração: um nível de interação entre objetos e um nível de comportamento interno dos objetos.

Um modelo RPOO é formado por um conjunto de classes e suas redes de Petri. As classes e seus relacionamentos com outras classes são descritas nos moldes convencionais da maioria dos modelos OO. Para cada classe, há exatamente uma rede de Petri correspondente no modelo RPOO. Desta maneira, uma classe e sua rede correspondente constituem um modelo formal para um conjunto de objetos. O comportamento dos objetos são descritos por redes de Petri coloridas modificadas com inscrições de interação.

Objetos podem interagir através de mensagens. Em um modelo RPOO, a troca de mensagens entre objetos se dá através de inscrições de interação anotadas junto às transições das redes que descrevem as classes. As inscrições descrevem ações que são efetuadas pelo objeto quando uma transição dispara. Os principais tipos de ações que podem ser associadas às transições são: (1) instanciação de uma classe, usada para criar objetos (x = new X); (2) entrada de dados – recebimento de uma mensagem por um objeto (*x?mensagem*); (3) saída síncrona de dados – envio de uma mensagem, de forma síncrona, de um objeto para outro (*x!mensagem*); (4) saída assíncrona de dados – envio de uma mensagem, de forma assíncrona, de um objeto para outro (*x.mensagem*); (5) desligamento, desfaz ligações entre objetos (-*x*); e (6) terminação da execução de um objeto (end). Uma inscrição pode conter várias ações e todas as ações de uma inscrição são executadas atomicamente quando a transição que contém a inscrição dispara.

Embora o comportamento de cada objeto seja descrito por sua rede de Petri, o comportamento de um modelo RPOO deve também considerar as interações e ligações no nível de objeto. Uma *configuração* em RPOO define um retrato do estado do sistema de objetos em um dado instante de seu funcionamento. Um configuração consiste do conjunto de objetos do sistema, as ligações entre estes objetos, o conjunto de mensagens pendentes trocadas por estes objetos, além dos estados internos das redes que modelam os objetos. A visão OO de uma configuração é definida como uma estrutura de objetos. O comportamento de um sistema RPOO pode ser observado a partir da definição de uma configuração inicial. Novas configurações são alcançadas a partir da configuração inicial e representam novos estados do sistema, resultado da ocorrência de ações descritas nos modelos dos objetos. Para cada tipo de ação definida para o sistema de objetos, existem regras que guiam a mudança de configuração de uma instância do modelo. Como exemplo de regra temos que um objeto não pode enviar uma mensagem para um outro objeto que ele não mantém uma ligação.

O espaço de estados de um modelo RPOO é um grafo que representa todas as configurações alcançáveis a partir da configuração inicial. Um nó representa uma

configuração e um arco ligando duas configurações indica a ação que foi executada para transformar uma configuração em outra.

2.2.1. Simulador de Sistemas de Objetos

O Simulador de Sistemas de Objetos (SSO) [Santos 2003] é uma ferramenta que permite simular as mudanças ocorridas em um sistema de objetos, através da execução de eventos. Um evento corresponde a uma ação ou conjunto de ações que ocorrem de forma atômica. Assim é possível exercitar a visão OO de um modelo RPOO, simulando diferentes situações de interação entre os objetos de um sistema. A análise efetuada sobre as estruturas de objetos alcançadas descreve o comportamento do sistema em termos da comunicação entre seus elementos, abstraindo detalhes do comportamento interno. A ferramenta, desenvolvida em Java, possui um conjunto de regras que determinam estados dos sistemas de objetos em função da aplicação de eventos a um determinado estado do sistema de objetos. A simulação é apresentada graficamente ao usuário. A ferramenta permite ainda a interação com o usuário através de linha de comando. Neste caso, o usuário utiliza uma forma textual para definir a estrutura inicial e as ações que serão executadas. Os cenários de simulação exercitados podem ser guardados em arquivos para serem usadas com outras ferramentas para diferentes propósitos.

2.2.2. JMobile

JMobile [Silva 2005] consiste de uma nova notação para RPOO e um framework que dá suporte a ferramentas de simulação de modelos RPOO e de geração de espaços de estados de modelos RPOO. Do ponto de vista da notação, JMobile utiliza uma sintaxe baseada na linguagem de programação Java para definir as inscrições dos modelos de Rede de Petri. Do ponto de vista do framework, JMobile disponibiliza em sua API, classes e métodos responsáveis por prover acessos às informações dos modelos e pela simulação destes modelos. A partir do framework JMobile, um conjunto de ferramentas (JMobile Tools) foi desenvolvido, incluindo um simulador e um gerador de espaço de estados para modelos RPOO. Diferente do SSO, o simulador considera além da visão OO o estado interno das redes que modelam os objetos. O gerador de espaço de estados, por sua vez, recebe como entrada um modelo RPOO e sua configuração inicial, gerando o espaço de estado correspondente.

3. Método para a Geração de Objetivos e Casos de Teste

Geração automática de casos de teste a partir de especificações formais pode aumentar a qualidade e a confiabilidade de processos de teste, visto que casos de teste tendem a ser consistentes, i.e., não rejeitam implementações corretas e podem ser convertidos diretamente em código, minimizando a introdução de defeitos de codificação no caso de serem voltados a execução automática. Conjuntos de teste podem ser mais precisamente avaliados com relação à métricas de cobertura de requisitos.

No entanto, alguns desafios ainda precisam ser vencidos. Especificações raramente são completas e com isto fica difícil assegurar que um software foi efetivamente testado apenas com base nos casos de teste gerados [Fernandez et al. 2004]. Além disso,

o processo é fortemente dependente da qualidade da especificação: ambigüidades e inconsistências são passadas diretamente para os casos de teste, invalidando todo o processo. Além de consistentes, conjuntos de caso de teste precisam refletir propriedades de interesse do sistema. Visto que não é possível garantir corretude com testes, a meta é detectar o maior número possível de defeitos relevantes. Métodos formais convencionais são voltados a dar suporte a verificação e, portanto, não apoiam diretamente as metas e atividades de um processo de teste. A escolha e direcionamento de casos de teste continua a ser uma atividade empírica onde diferentes fatores não formalizáveis vão nortear a escolha dos melhores casos de teste.

Neste cenário, os principais requisitos para um método prático de teste formal são [Tretmans 1999, Jard and Jéron 2004]:

- Geração de conjuntos de casos de teste consistentes. Isto é diretamente influenciado pelos formalismos e algoritmos de geração adotados que devem seguir uma boa fundamentação teórica e exigir um mínimo de trabalho manual;
- Geração de conjuntos de casos de teste válidos, i.e., que reflitam propriedades desejadas por seus usuários finais. Isto é diretamente influenciado pela qualidade dos processos de verificação e validação das especificações;
- Geração a partir de especificações parciais, visto que a geração total é usualmente impossível e tem pouca utilidade, pois os conjuntos de casos de teste tendem a ser muito grandes e repetitivos. Além disto, especificações completas geralmente não estão disponíveis;
- Seleção de casos de teste com base em propriedades ou funcionalidades específicas de interesse. Isto é fortemente dependente do uso de notações comumente adotadas e da disponibilidade de visões abstratas que tornem possível a investigação de funcionalidades e propriedades do sistema.
- A necessidade de construção de novos artefatos deve ser minimizada, pois o processo de construção de uma especificação formal já é por si de altos custos. Para tal, é desejável que modelos de teste sejam derivados diretamente dos construídos em processos de desenvolvimento existentes.

Apesar de aspectos teóricos de teste formal já estarem bastante consolidados e ferramentas de suporte já existirem, as abordagens atuais ainda não cobrem todos estes pontos em conjunto de forma satisfatória. Visando atingi-los, um método de teste formal é proposto neste artigo. As principais características são:

- Um mesmo modelo é usado no processo de verificação de modelos e para a geração de casos de teste, a fim de garantir que este modelo será devidamente verificado e validado;
- A geração de casos de teste é feita com base em algoritmos baseados em teoria consolidada de teste, garantindo a geração de conjuntos de casos de teste consistentes;
- A geração é dirigida por objetivos de teste, tornando possível o enfoque em partes específicas do modelo;
- Objetivos de teste são definidos com base em abstrações do modelo de teste e podem ser validados.

Para tal, RPOO foi adotado com formalismo base. As diferentes visões de RPOO e o conjunto de ferramentas já disponível e sua fácil integração com geradores de casos

de teste como TGV foram determinantes nesta escolha. Objetivos de teste podem ser gerados em um processo intuitivo por projetista de teste a partir da simulação do modelo de objetos, onde são considerados apenas conceitos de orientação a objetos referentes a criação de objetos e passagem de mensagens. Cada objetivo é representado por uma seqüência válida de eventos a partir de uma certa configuração do sistema.

A estrutura geral do método é apresentada na Figura 1, onde são ilustradas as principais entradas/saídas produzidas e as ferramentas adotadas. Os sistemas a serem testados precisam estar modelados em RPOO. Assumimos que estes modelos foram verificados usando ferramentas para verificação de modelos RPOO tal como a ferramenta Veritas [Rodrigues et al. 2004]. Utilizamos JMobile (Seção 2) para dar suporte a geração de espaços de estados que é o modelo a partir do qual casos de teste são gerados.

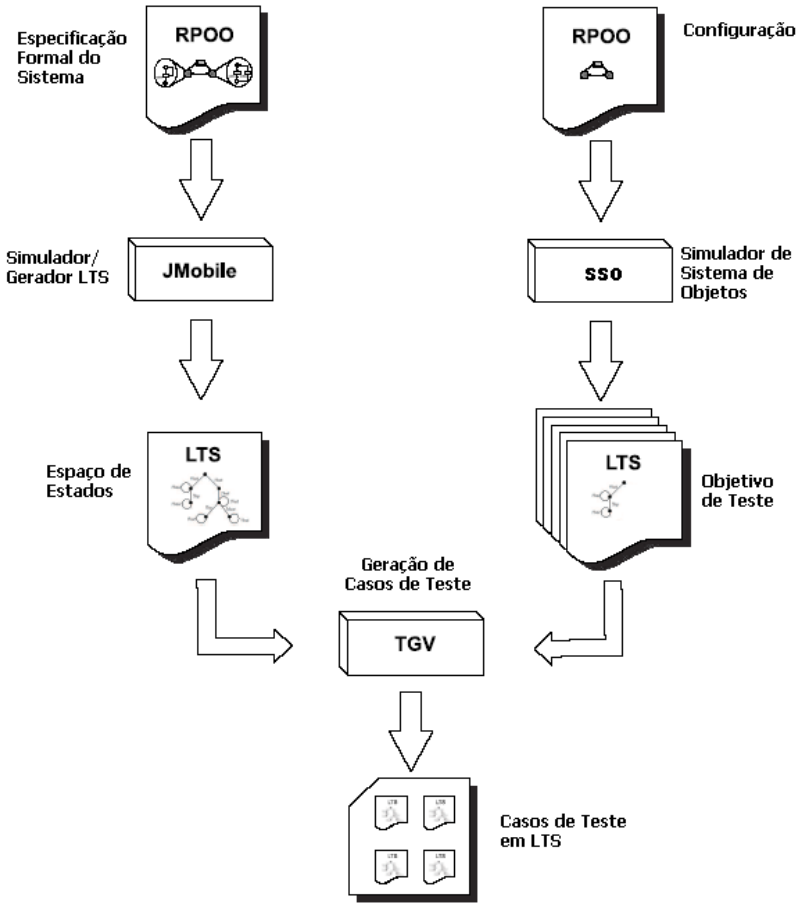


Figura 1. Visão Geral do Método para Geração Automática de Casos de Teste

JMobile recebe o modelo RPOO como entrada e gera o seu respectivo espaço de estados em dois tipos de formatos. Um utilizado pelo verificador de modelos Veritas e outro utilizado pela ferramenta Aldebaran [Fernandez 1989]. Este último formato é utilizado no nosso método, já que a ferramenta de geração de casos de teste TGV recebe suas entradas em tal formato.

O objetivo de teste (parte direita da Figura), que constitui um comportamento específico da aplicação que se deseja testar, é obtido automaticamente através de simulações. Para isto utilizamos a ferramenta SSO (Seção 2). SSO permite simular as

alterações ocorridas no modelo da aplicação dado uma configuração, gerando possíveis objetivos de teste como seqüências de eventos. As entradas para esta ferramenta são: (i) a configuração de um sistema de objetos, que é formada pelos objetos que compõem o sistema e suas relações; e, (ii) o conjunto de eventos da execução a qual se pretende simular. A saída também é processada pela ferramenta Aldebaran a fim de produzir um IOLTS que é dado como entrada para TGV, para cada objetivo de teste.

Grande parte das ferramentas de geração de casos de teste recebe como entrada o sistema especificado em LTS e aplica sua técnica através de seus algoritmos. No nosso método, a ferramenta TGV é utilizada para a geração dos casos de teste propriamente dita. Esta ferramenta recebe como entrada o modelo do sistema e a especificação de um objetivo de teste, e seleciona os casos de teste a partir da especificação com base em tal objetivo de teste. O algoritmo utilizado por esta ferramenta é o de busca em profundidade (Depth First Search - DFS). Os casos de teste produzidos são apresentados, também, em LTS e poderão ser implementados ou convertidos para outra linguagem como, por exemplo, *message sequence chart* e diagrama de seqüência de UML.

O uso de TGV e o fato de que os objetivos de teste são seqüências válidas da especificação garantem que o conjunto de casos de teste gerado é *consistente*, segundo [Tretmans 1999], e que os objetivos de teste são *e-completos*, segundo [de Vries and Tretmans 2001], i.e., os casos de teste gerados a partir destes objetivos conseguem distinguir implementações que exibem e que não exibem o comportamento que se deseja observar.

4. Estudo de Caso

Esta seção mostra uma aplicação do método proposto na Seção 3. O objetivo deste estudo de caso é o de apresentar um exemplo prático do uso do método, mostrando sua aplicabilidade e apresentando indícios reais de sua viabilidade.

4.1. Visão Geral

Trata-se de um sistema de apoio às atividades de comitês de programa em conferências, que será chamado simplesmente de sistema de conferências. A aplicação gerencia atividades de comitês de programas de conferências, tais como submissão de artigos, processo de avaliação e notificação de aceitação ou rejeição de artigos aos autores.

Esta aplicação foi desenvolvida sobre o paradigma de Agentes Móveis [Fuggeta et al. 1998]. Agentes Móveis é um paradigma para o desenvolvimento de sistemas distribuídos que surgiu com o intuito de unir os conceitos oriundos dos agentes de software com os presentes em mobilidade de código. Dado que os sistemas de tal paradigma são formados por agentes, podemos dizer que teremos entidades atuando sobre o interesse de uma outra entidade, com características como autonomia, inteligência e cooperação. Já com relação ao termo "móvel", temos que os agentes são capazes de alterar, dinamicamente, suas ligações com seu lugar de execução.

4.2. Modelos RPOO

Os modelos RPOO completos, contendo todos os diagramas, redes de Petri e suas respectivas explicações podem ser encontrados na dissertação de Figueiredo [Figueiredo 2005]. No geral, agentes e agências são modelados como objetos que se relacionam e trocam mensagens entre si. As principais entidades envolvidas neste estudo de caso são:

- **Agente Coordenador**: Agente estacionário responsável por prover interface gráfica com o coordenador do membro de comitê e, ao receber a solicitação de revisão de um artigo, cria um agente **Agente Formulário Revisão**, delegando a responsabilidade de obter as revisões para os artigos.
- **Agente Formulário Revisão**: Responsável por obter uma revisão para um determinado artigo. É um agente móvel que migra por agências de revisores em busca de revisões e, ao final, registra essa revisão junto ao **Agente Coordenador**.
- **Agência Coordenador**: Agência onde estará executando o **AgenteCoordenador** e onde o coordenador do comitê de programa estará localizado.
- **Agência Membro Comitê**: Agência por onde o **Agente Formulário Revisão** irá passar e onde estará situado o membro do comitê responsável por um artigo.
- **Agência Revisor**: Agência por onde o **Agente Formulário Revisão** irá passar e onde estará situado um possível revisor de um artigo.

Para cada classe do modelo, temos uma rede de Petri que descreve o comportamento de seus objetos.

4.3. Geração e Seleção de Objetivos de Teste

De acordo com o método proposto, quaisquer seqüências de eventos geradas por SSO podem ser utilizadas como objetivos de teste que denotam um comportamento de interesse que se quer testar no sistema. Em particular, neste estudo de caso, optamos por testar o comportamento esperado para padrões de projeto para agentes móveis que foram adotados no desenvolvimento da aplicação. Desta forma, SSO foi utilizado para simular o comportamento dos padrões e as seqüências de eventos obtidas foram utilizadas para definir os objetivos de teste.

Diferentes padrões de projeto foram considerados para obter objetivos de teste e, conseqüentemente, obter casos de teste para o sistema apresentado acima. Neste artigo, iremos nos ater ao padrão *Itinerary*. Este padrão define uma forma na qual um agente pode migrar por diferentes agências de um sistema executando uma tarefa específica em cada uma delas. A solução apresentada pelo padrão descreve a infra-estrutura necessária para que este agente (chamado *ItineraryAgent*) migre por tais agências executando esta tarefa em cada uma delas e retorne, ao final, à agência de origem com o resultado da execução.

Este padrão é implementado pelo agente *AgenteFormRevisao* quando este percorre uma lista de agências (seu itinerário) cujos revisores requereram a aprovação do formulário de revisão, onde a tarefa a ser executada é a de aprovar as revisões contidas no formulário. A Figura 2 apresenta um objetivo de teste obtido para o padrão *Itinerary*. Para realizar a simulação que propiciou sua geração, foi necessário conhecer que o padrão é utilizado no sistema a partir do momento em que um *AgenteFormRevisao* finaliza a sua revisão (ação .***agenteForm.***\\.finalizarRevisao(.**).**), seguindo o seqüenciamento de mensagens apresentado, até que o *AgenteCoordenador* recebe uma mensagem de registrar resultado de revisão (ação .***agenteCoord.***\\.registrarResultado(.**).**). Além disto, foi necessário ter o conhecimento das ações que ocorrem bem como suas ordens. Note que este é um grafo abstrato e que só as ações de interesse estão no grafo, deixando as demais abstraídas nas transições com rótulo *.

Pela Figura 2, vemos que o objetivo de teste seleciona a parte do sistema que se inicia a partir do momento em que um revisor finaliza a sua revisão (vide transição

*.*agenteForm.*\.finalizarRevisao(.*).** que parte do estado *0* para o estado *1*) e então começa o processo de aprovação, que é o trecho do sistema que o padrão *Itinerary* é implementado. Além disso, podemos ver que o objetivo de teste seleciona as linhas de execução em que há pelo menos uma agência no itinerário do agente. Isto é obtido no momento em que pelo menos uma transição *.*gui.*\.showTelaAprovar(.*).** precisa ser executada (transição que parte do estado *2* para o estado *3*), ou seja, há pelo menos uma agência solicitando aprovação.

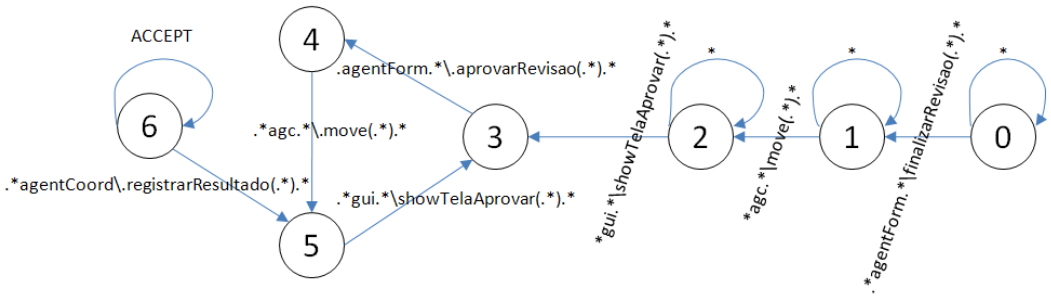


Figura 2. LTS do Objetivo de Teste para o Padrão de Projeto *Itinerary*

4.4. Resultados da Geração dos Casos de Teste

A geração de casos de teste foi realizada a partir da especificação do sistema (espaço de estados) e com base nos objetivos de teste elaborados. Tal geração se deu com o uso da ferramenta TGV, onde o espaço de estados e os objetivos de teste foram entradas do processo, e um grafo foi produzido contendo os casos de teste para cada objetivo de teste. Estes grafos produzidos contêm todos os casos de teste para cada objetivo de teste.

A geração dos casos de teste com TGV é simples, já que consiste em fornecer um LTS do sistema a ser testado e um objetivo de teste como entradas, ambos no formato Aldebaran. A ferramenta retorna um outro LTS, chamado de *Complete Test Graph*, contendo os casos de teste gerados. Uma vez que este grafo gerado é relativamente grande, sua visualização através de uma figura fica inviável. Desta forma, a Tabela 1 apresenta algumas informações para cada grafo gerado a partir de alguns objetivos de teste.

Objetivo de Teste	Qtd. de Estados	Qtd. de Transições	Tempo de Geração
Padrão Itinerary	30.381	69.452	8 min. e 23 seg.
Padrão MasterSlave	17.180	43.608	8 min. e 8 seg.
Padrão Barnching	16.954	43.201	9 min. e 28 seg.

Tabela 1. Dados sobre a Geração dos Casos de Teste por Objetivo de Teste

Como foi dito, de cada grafo de teste gerado, alguns casos de teste foram selecionados para serem implementados e executados sobre a aplicação. Uma vez que o intuito deste trabalho é o de apresentar a viabilidade de implementação dos casos de teste gerados pelo método e de mostrar a eficiência dos mesmos em revelar defeitos nas aplicações, selecionamos um conjunto de casos de teste para mostrar sua implementação e execução, mostrando os defeitos encontrados, quando for o caso. A seleção desses casos de teste foi feita por meio de uma simples seleção aleatória de caminhos dos grafos de teste gerados. Estes caminhos possuem como estado inicial o estado inicial do grafo de teste e como estado final um estado de aceitação.

Os casos de teste são seqüências de entradas e saídas do sistema. O testador fornecerá as entradas ao sistema a ser testado e é esperado que este retorne as saídas definidas. A cada rótulo de transição do caso de teste, a ferramenta acrescenta a palavra *INPUT* ou a palavra *OUTPUT*. A transição que contém a palavra *INPUT* indica ao testador que este deve esperar aquela ação como uma entrada, ou seja, uma saída do sistema a ser testado. Já a transição que possui a palavra *OUTPUT* indica ao testador que este deve produzir aquela ação para que sirva de entrada para o sistema a ser testado. Note que os termos *INPUT* e *OUTPUT* se referem ao programa que irá testar a aplicação, comumente chamado de *test driver*. A Figura 3 apresenta um trecho de um determinado caso de teste para o sistema de conferências. Como pode ser visto na figura, a primeira ação (linha 2) do caso de teste é o envio de uma mensagem do *AgenteCoordenador* para a sua interface gráfica solicitando que uma tela seja mostrada. A palavra *INPUT* informa que esta é uma mensagem que deve ser observada pelo testador, é uma entrada para quem irá testar o sistema. A seqüência de ações procede até que, na linha 25, o *agentFormOriginal* solicita que seja migrado para a agência *agcCoord*.

```
 ------------------------------------------------------------------------
 1 <initial state>
 2 "agentCoord:guicoord.show(); INPUT"
 3 "guicoord:agentCoord.gerarFormRevisao((’agcMemb’,1)); OUTPUT"
 4 "agentCoord:conf.getDadosConferencia() &
    agentCoord:conf.criarRegistroRevisao((’agcMemb’,1)); INPUT"
 5 "conf:agentCoord.dadosConferencia(’dadosconf1’); OUTPUT"
 6 "conf:agentCoord.registroRevisao((’agcMemb’,1)); OUTPUT"
 .
 .
 .
23 "agentFormOriginal:
    guirevisao-agentFormOriginal.showTelaAprovar(); INPUT"
24 "guirevisao-agentFormOriginal:
    agentFormOriginal.aprovarRevisao(); OUTPUT"
25 "agentFormOriginal:agcMemb.move(’agcCoord’); INPUT (PASS)"
26 <goal state>
 ------------------------------------------------------------------------
```

Figura 3. Trecho de um Caso de Teste para o Sistema de Conferências

4.5. Implementação e Execução dos Casos de Teste

A abordagem escolhida para a implementação dos casos de teste foi simples, onde, basicamente, cada ponto de controle foi mapeado em uma entrada para a implementação e cada ponto de observação para uma saída da implementação. Tanto os pontos de controle como os de observação geraram instrumentações no código que visavam gerar linhas de execução que pudessem ser comparadas com os casos de teste gerados. Desta forma e após a execução de cada caso de teste, um arquivo contendo as ações executadas é criado (arquivo de *log*). Este foi usado para avaliar as execuções do sistema, pois as ações ali contidas devem ser as mesmas esperadas pelo caso de teste, para uma execução correta.

A execução dos casos de teste gerados foi feita de forma manual devido a falta de infra-estrutura para execução automática de testes na plataforma de agentes móveis escolhida. As ações descritas pelos mesmos foram entradas no sistema e suas saídas foram observadas. Na prática, para cada caso de teste, o sistema foi exercitado com as entradas previstas nos casos de teste e, com o arquivo de *log* gerado pelo *Test Driver*, a execução foi avaliada em correta ou incorreta em relação à especificação. Na Tabela 2, são apresentados os casos de teste para o objetivo de teste (gerado a partir do padrão *Itinerary*) exercitados e os resultados das execuções. Apresentamos uma breve descrição dos casos de teste e mostramos os resultados da execução, através dos defeitos encontrados, caso

Objetivo de Teste para o Padrão *Itinerary*	
Caso de Teste 01	
- A quantidade de revisões solicitadas é igual a um (um agente *AgenteFormRevisao* será criado). - Membro do comitê o envia à agência *agcRev1* e solicita o seu retorno. - Revisor reenvia o formulário à agência *agcRev2* sem solicitar o seu retorno. - Revisor entra com dados de revisão e o finaliza.	
Resultados:	O sistema se comportou em conformidade com a especificação.
Caso de Teste 02	
- A quantidade de revisões solicitadas é igual a um (um agente *AgenteFormRevisao* será criado). - Membro do comitê o envia à agência *agcRev1* e solicita o seu retorno. - Revisor reenvia o formulário à agência *agcRev1* (mesma agência em que se encontra) solicitando o seu retorno. - Revisor entra com dados de revisão e o finaliza.	
Resultados:	O agente não consegue migrar para a agência *agcRev1* em sua segunda tentativa, dando indícios de que há um defeito no tratamento de quando o agente solicita uma migração para a mesma agência em que está localizado.
Caso de Teste 03	
- A quantidade de revisões solicitadas é igual a dois (dois agentes *AgenteFormRevisao* serão criados). - Apenas o agente original é utilizado. - Membro do comitê o envia à agência *agcRev1* e solicita o seu retorno. - Revisor reenvia o formulário à agência *agcRev2* solicitando o seu retorno. - Revisor entra com dados de revisão e o finaliza. - Agente retorna à agência *agcRev1* e o revisor aprova a revisão. - Agente retorna à agência do membro de comitê e o membro aprova a revisão.	
Resultados:	O agente não migra para a agência *agcRev1* para a aprovação, apenas para a agência do membro de comitê. Uma vez que a lista de agências que solicitaram a revisão é implementada pelo padrão *Itinerary*, encontramos fortes indícios de que o padrão não foi implementado de forma correta.

Tabela 2. Casos de Teste Extraídos do Objetivo de Teste para o Padrão *Itinerary*

existam. Os casos de teste e uma descrição mais completa podem ser encontrados em [Figueiredo 2005].

A execução dos casos de teste nos permitiu avaliar o potencial destes em revelar defeitos. Especificamente em relação à característica de mobilidade, os casos de teste se mostraram úteis em revelar defeitos na implementação desta característica, como pode ser visto nos resultados dos Casos de Teste 02 e 03 do padrão *Itinerary* (Tabela 2). Assim, os objetivos de teste se mostraram interessantes quando se deseja testar partes específicas de um sistema, já que seus casos de teste detectaram defeitos com relação ao comportamento especificado pelos objetivos ou mostraram conformidade de tal comportamento.

5. Trabalhos Relacionados

Geração de casos de teste usando verificadores de modelo tem sido bastante explorada na comunidade [de Vries and Tretmans 1998, Jard and Jéron 2004, Bonifácio et al. 2006]. Casos de teste são gerados a partir de caminhos no modelo e objetivos de teste são formalizados, usualmente, usando lógica temporal [Fernandez et al. 2004, da Silva and Machado 2006], LTSs [Jard and Jéron 2004], diagramas de estado UML [Hartman and Nagin 2004], e Message Sequence Charts (MSC) [Henniger et al. 2003]. Abordagens simbólicas também têm sido propostas [Clarke et al. 2002, Frantzen et al. 2005].

A contribuição original deste trabalho é apresentar um método onde casos de teste são gerados com base em objetivos de teste que são definidos em um nível de abstração alto e também podem ser gerados automaticamente. Tal nível de abstração é fundamental para que o testador possa expressar propriedades de forma prática e intuitiva, através de seqüências de ativação de objetos. Estas seqüências podem representar instâncias de interesse do comportamento de sistemas. E podem ser reaproveitadas diretamente do processo de simulação dos modelos, sendo geradas automaticamente por ferramentas, como SSO. Em abordagens como as propostas por [Hartman and Nagin 2004] e [Henniger et al. 2003], objetivos de teste são construídos como um novo artefato independente. Já as propostas por [Fernandez et al. 2004, da Silva and Machado 2006] dão enfoque ao teste de propriedades temporais, podendo ser usadas de forma complementar

a proposta deste artigo.

6. Considerações Finais

Este artigo apresenta um método para a geração e seleção automática de casos de teste baseadas em objetivos de teste para sistemas reativos. O método é baseado em fundamentação teórica sólida e apoiado pelo uso de ferramentas. Outra característica importante é a sua integração com práticas de verificação de modelos, fazendo reuso de artefatos gerados por estas práticas. A proposta é inovadora no sentido de que objetivos de teste também podem ser gerados automaticamente. O estudo de caso apresentado mostra uma aplicação onde os objetivos de teste representam comportamentos desejados na implementação de padrões de projeto. Este comportamento pode ser obtido a partir do uso da ferramenta SSO. A ferramenta faz um log da simulação do envio de uma seqüência de eventos válida sobre o modelo de objetos e, ao final, esta seqüência é exportada como um LTS representando, no caso, o objetivo de teste desejado. Com isto, temos ganhos em confiabilidade na definição destes objetivos: são precisos e fiéis ao modelo. Temos também ganhos com versatilidade, pois o projetista de teste tem uma forma mais intuitiva para defini-los, através da animação do modelo usando SSO. Diagramas de classe RPOO são similares a diagramas de classe em UML padrão. Neste sentido, o método pode ser aplicado por testadores com conhecimento em UML e orientação a objetos, abstraindo o detalhamento das classes em modelos de redes de Petri.

Como trabalhos futuros, o método será estendido a fim de considerar padrões de teste na definição de objetivos de teste. Extensões e variações do formalismo RPOO serão consideradas a fim de ampliar o escopo de uso do método para software baseado em componentes e com restrições temporais.

Referências

Bonifácio, A., Moura, A., Simão, A., and Maldonado, J. (2006). Conformance testing using timed extended finite state machines and model checking. In *Proceedings of Brazilian Symposium on Formal Methods*.

Cartaxo, E. G., Andrade, W. L., Neto, F. G. O., and Machado, P. D. L. (2008). LTS-BT: A tool to generate and select functional test cases for embedded systems. In *Proceedings of 23rd Annual ACM Symposium on Applied Computing*, volume 2, pages 1540–1544.

Clarke, D., Jeron, T., Rusu, V., and Zinovieva, E. (2002). STG – A Symbolic Test Generation Tool. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*. Springer.

Clarke, E. M., Grumberg, O., and Peled, D. (1999). *Model Checking*. MIT Press.

da Silva, D. A. and Machado, P. D. L. (2006). Towards test purpose generation from CTL properties for reactive systems. *Electr. Notes Theor. Comput. Sci.*, 164(4):29–40.

de Vries, R. G. and Tretmans, J. (1998). On-the-fly conformance testing using spin. In *Proceedings of Fourth Workshop on Automata Theoretic Verification with the Spin Model Checker*, pages 115–128.

de Vries, R. G. and Tretmans, J. (2001). Towards formal test purposes. In *Proceedings of 1st International Workshop on Formal Approaches to Testing of Software (FATES)*, pages 61–76, Aalborg, Denmark.

Fernandez, J., Mounier, L., and Pachon, C. (2004). Property oriented test case generation. In *Proceedings of Formal Approaches to Software Testing*, volume 2931 of *LNCS*, pages 147–163. Springer.

Fernandez, J.-C. (1989). Aldebaran: A Tool for Verification of Communicating Processes. Technical report, Rapport SPECTRE, C14, Laboratoire de Génie Informatique - Institut IMAG, Grenoble - França.

Fernandez, J.-C., Jard, C., Jeron, T., and Viho, C. (1996). Using on-the-fly Verification Techniques for the Generation of Test Suites. In *Proc. of the* 8^{th} *Int. Conf. on Computer Aided Verification CAV*, volume 1102 of *LNCS*, pages 348–359. Springer.

Figueiredo, A. L. L. (2005). Geração automática de casos de teste para sistemas baseados em agentes móveis. Master's thesis, COPIN/Universidade Federal de Campina Grande. http://www.dsc.ufcg.edu.br/ andrel/arquivos/dissertacao.pdf.

Frantzen, L., Tretmans, J., and Willemse, T. A. C. (2005). Test generation based on symbolic specifications. In Grabowski, J. and Nielsen, B., editors, *Proceedings of FATES'04*, volume 3395 of *LNCS*, pages 1–15. Springer.

Fuggeta, A., Picco, G., and Vigna, G. (1998). Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24:342–361.

Gaudel, M.-C. (1995). Testing can be formal, too. In *Proceedings of TAPSOFT*, LNCS, pages 82–96, London, UK. Springer.

Guerrero, D. D. S. (2002). *Redes de Petri Orientadas a Objetos*. PhD thesis, COPELE/Universidade Federal de Campina Grande.

Hartman, A. and Nagin, K. (2004). The AGEDIS tools for model based testing. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 129–132, New York, NY, USA. ACM Press.

Henniger, O., Lu, M., and Ural, H. (2003). Automatic generation of test purposes for testing distributed systems. In *Formal Approaches to Software Testing, Proceedings of FATES'03*, volume 2931 of *LNCS*, pages 178–191. Springer.

Jard, C. and Jéron, T. (2004). Tgv: theory, principles and algorithms, a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Software Tools for Technology Transfer (STTT)*, 6.

Jorgensen, P. C. (2002). *Software Testing - A Craftsman's Approach*. CRC Press.

Rodrigues, C. L., Barbosa, P. E. S., Guerrero, D. D. S., and de Figueiredo, J. C. A. (2004). Rpoo model checker. In *Anais do Simpósio Brasileiro de Engenharia de Software - Sessão de Ferramentas*, Brasília - Brasil.

Santos, J. A. M. (2003). Suporte à análise e verificação de modelos rpoo. Master's thesis, COPIN/Universidade Federal de Campina Grande, UFCG.

Silva, T. M. (2005). Simulação automática e geração de espaço de estados de modelos em redes de petri orientadas a objetos. Master's thesis, COPIN/Universidade Federal de Campina Grande.

Tretmans, J. (1999). Testing concurrent systems: A formal approach. In *Proceedings of CONCUR'99*, volume 1664 of *LNCS*, pages 46–65. Springer.

Aplicação de algoritmos evolutivos na geração automática de dados de teste de conformidade

Thaise Yano[1], **Eliane Martins**[1], **Fabiano Luís de Sousa**[2]

[1]Instituto de Computação – Universidade Estadual de Campinas (Unicamp)
Caixa Postal 6176 – 13083-970 – Campinas – SP – Brasil

[2]Instituto Nacional de Pesquisas Espaciais (INPE)
São José dos Campos – SP – Brasil

tyano@ic.unicamp.br, eliane@ic.unicamp.br, fabiano@dem.inpe.br

Abstract. *In order to reduce the cost and effort in the software development, the automatic test data generation have been considered as an important activity to aid the testing process. Recently, there has been a great interest in search based testing that apply optimization techniques in the test data generation. In this context, several works can be found that use evolutionary algorithms to structural testing, however few of them focus on the functional testing. Then in this paper, an initial assessment is done to check the efficacy of GEO algorithm, a recent evolutionary algorithm, in dealing with data generation for conformance testing.*

Resumo. *Com o objetivo de reduzir custo e esforço no desenvolvimento de software, a geração automática de dados de teste tem sido considerada como uma importante atividade para auxiliar o processo de teste. Recentemente, houve um grande interesse sobre teste baseado em busca que procura aplicar técnicas de otimização na geração de dados de teste. Nesse contexto, pode-se encontrar diversos trabalhos que utilizam algoritmos evolutivos para o teste estrutural, porém poucos abordam o teste funcional. Dessa maneira, neste artigo é feita uma avaliação inicial da eficácia do algoritmo GEO, um algoritmo evolutivo recente, para gerar dados de teste de conformidade.*

1. Introdução

A atividade de teste tem sido apontada como uma das mais onerosas no desenvolvimento de software [Harrold 2000]. Um relatório preparado para o NIST (*National Institute of Standards and Technology*) [Gallaher and Kropp 2002] quantifica os altos impactos econômicos de uma infra-estrutura de teste de software inadequada e como melhoria cita, por exemplo, a geração automática de teste.

Bertolino [Bertolino 2007] descreve os desafios mais relevantes na pesquisa de teste de software, entre eles a geração de dados de teste. E nesse contexto, comenta a promissora relação entre o teste de software e outras áreas de pesquisa, tal como o uso de abordagens baseada em busca [McMinn 2004, Harman 2007] na geração de dados de teste. Outro desafio citado é o teste baseado em modelos, sendo que a idéia principal é usar modelos no desenvolvimento de software para auxiliar o processo de teste, particularmente, na geração automática de casos de teste.

Dessa maneira, um dos problemas importantes na atividade de teste de software é o desenvolvimento de dados de teste, que consiste em identificar dados de entrada que satisfaçam um determinado critério de teste. Automatizar a geração de dados de teste contribuiria para a redução de custo e esforços ao processo de desenvolvimento de software. Nesse sentido, as principais abordagens para a geração automática de dados de teste são [Michael et al. 2001]: aleatória, simbólica e dinâmica. No teste aleatório, dados de teste são selecionados aleatoriamente do domínio de entrada, mas a probabilidade de se selecionar dados que pertençam a uma pequena porcentagem do domínio de entrada é muito baixa. Por outro lado, é comumente utilizado na literatura como uma medida de comparação por ser um método simples e de fácil implementação. As abordagens que utilizam a execução simbólica [Offutt 1991] atribuem valores simbólicos às variáveis, ao invés de usar seus reais valores, e reduzem o problema de geração de dados de teste à resolução de expressões algébricas que envolvem um conjunto de restrições de igualdade e desigualdade sobre esses valores simbólicos. Uma desvantagem é que esse método apresenta problemas com *loops*, vetores e referências a ponteiros que limitam seu uso nos dias de hoje [Michael et al. 2001, Korel 1990]. Já a abordagem dinâmica de geração de dados de teste baseia-se na execução dos valores reais das variáveis. Quando um requisito de teste não é satisfeito, uma função objetivo é associada a ele e métodos de otimização de funções são utilizados para buscar dados que mais se aproximam de satisfazer o requisito. Com base nessas informações, os dados de teste são incrementalmente modificados até que um deles satisfaça o requisito. Assim a geração de teste é reformulada como um problema de otimização. Como métodos de otimização utilizados nessa abordagem incluem o recozimento simulado [Tracey et al. 1998], método do gradiente [Korel 1990], *hill climbing* [Harman and McMinn 2007] e algoritmos evolutivos [McMinn 2004].

A área de pesquisa que investiga o uso de algoritmos evolutivos na atividade de teste é denominada como testes evolutivos. Nos últimos anos houve um grande crescimento no interesse por essa área, sendo aplicada para diferentes técnicas e critérios de teste. Muitos trabalhos focam o teste estrutural [Abreu 2006, Harman and McMinn 2007, Michael et al. 2001, Pargas et al. 1999, Watkins and Hufnagel 2006], mas também abordam com menor intensidade o teste funcional [Derderian et al. 2006, Guo et al. 2005, Jones et al. 1995, Tracey et al. 1998]. Entre os algoritmos evolutivos mais utilizados na geração de dados de teste está o algoritmo genético (AG). Porém outros algoritmos têm sido propostos para o teste evolutivo que apresentam a vantagem de possuir menos parâmetros a serem ajustados que o AG, tal como o GEO (*Generalized Extremal Optimization*) [DeSousa 2002]. Isso facilita o processo de configuração do algoritmo para obter um melhor desempenho. O GEO foi competitivo com outros métodos estocásticos em funções-teste [DeSousa et al. 2003a] e tem sido aplicado com sucesso em problemas reais de projeto ótimo [Galski et al. 2007, DeSousa et al. 2003a, DeSousa et al. 2003b, DeSousa et al. 2004]. No trabalho de Abreu [Abreu 2006] foi explorado pela primeira vez o uso do GEO em uma atividade da engenharia de software. Os estudos de casos mostraram que o GEO também é competitivo com o algoritmo genético na geração de dados de teste estrutural, exigindo menos esforço computacional para tal. Isso motivou a aplicação do GEO como uma opção interessante a ser utilizada na geração de dados de testes funcionais. Então neste trabalho é realizada uma investigação inicial do algoritmo GEO no teste de especificações baseadas em modelos, particularmente, Máquina de Estados Finitos (MEF). A abordagem é comparada com o teste aleatório.

O artigo está organizado da seguinte forma. Na Seção 2 é descrito o teste baseado em modelo e é apresentada a MEF. Na Seção 3 é apresentado o conceito de testes evolutivos. Na Seção 4 é descrito o algoritmo GEO. Na Seção 5 é mostrada a proposta de aplicação do GEO no contexto de teste de conformidade. Finalmente na Seção 6 conclui-se o trabalho e são apresentados os trabalhos futuros.

2. Teste baseado em modelo

O teste de conformidade é um teste baseado em modelo que tem sido aplicado com o objetivo de investigar se a implementação se comporta de acordo com a especificação, buscando detectar discrepâncias entre uma especificação e suas implementações. O teste de conformidade baseia-se principalmente nas abordagens de caixa-preta. Então os casos de testes são gerados com base na especificação e são utilizados no teste da implementação. Portanto os testes são dependentes do formalismo da especificação. O uso de uma especificação descrita em linguagem natural passa a ilusão de ser facilmente entendida, porém induz especificações longas e informais que geralmente contêm ambigüidades e dificultam a verificação de completitude e consistência [Bochmann and Petrenko 1994]. Assim justifica-se o uso de métodos formais, tal como as MEFs, para especificação de sistemas que requerem alto grau de confiabilidade e segurança.

Geurts et al. [Geurts et al. 1998] relatam um estudo de caso em que se obtiveram benefícios na fase de teste com o uso de métodos formais. Neste estudo de caso, devido à precisão, consistência e completitude das especificações formais, casos de teste eficientes e efetivos puderam ser sistematicamente derivados dessas especificações.

2.1. MEF

Uma Máquina de Estados Finitos (MEF) [Gill 1962] é uma máquina hipotética composta por estados e transições. A cada instante, a máquina pode estar em somente um de seus estados. Em resposta a um evento de entrada, a máquina gera um evento de saída e muda de estado. Tanto o evento de saída gerado quanto o novo estado são definidos unicamente em função do estado atual e do evento de entrada [Davis 1988].

Uma MEF pode ser definida como $M = (S, s_0, I, O, T)$ em que: S é um conjunto finito e não vazio de estados; $s_0 \in S$ é o estado inicial; I é um conjunto finito de entradas; O é um conjunto finito de saídas; e $T : D_s \rightarrow S \times O$ é a função de transição, onde $D_s \subseteq S \times I$ é o domínio da especificação. A função de transição representa as possíveis transições da máquina. A expressão $t(s_i, x) = (s_n, y)$ representa a transição do estado s_i para o estado s_n quando o símbolo de entrada x é recebido, produzindo a saída y.

Quando $D_s \subset S \times I$, T não é definida para cada par (s, x) e representam-se máquinas parcialmente especificadas. Para tais máquinas, uma suposição de completitude deve ser adotada a fim de especificar seu comportamento para entradas inesperadas. Por exemplo, pode-se ignorar as entradas inesperadas produzindo uma saída nula e permanecer no estado atual. Outra propriedade das máquinas é o determinismo, no qual para cada entrada existe no máximo uma transição definida em cada estado da máquina.

Uma MEF pode ser representada por um grafo direcionado. Um grafo $G(V, E)$ é uma estrutura definida por dois conjuntos não vazios: nós (V) e arestas (E). Um par ordenado de nós (v_i, v_n) de um grafo direcionado descreve uma aresta $e_k \in E$ que parte de v_i e termina em v_n. Um caminho em $G(V, E)$ é uma seqüência não vazia de aresta. Se

o nó inicial e final de um caminho são os mesmos, então o caminho forma um *loop*. Em particular, os nós representam os estados e as arestas representam as transições da MEF.

2.2. Geração de teste

O teste baseado em MEF é realizado usando-se seqüências de teste que consistem de uma seqüência de entradas (e eventualmente das saídas correspondentes) a partir do estado inicial da máquina.

Os métodos de geração de casos de teste que exploram o aspecto de controle das MEFs procuram encontrar falhas de saída e de transferência na implementação em teste. Vários métodos para a geração de seqüências de teste de MEF são encontrados na literatura, tais como [Bourhfir et al. 1996]: DS - *Distinguishing Sequence*, Método W, UIO - *Unique-Input-Output* e Método Wp. Tais métodos baseiam-se em propriedades e seqüências especiais das máquinas para a determinação de um conjunto de casos de teste que seja pequeno e, ao mesmo tempo, consiga a cobertura do maior número possível de falhas contidas em uma máquina [Fujiwara et al. 1991]. As propriedades das máquinas que são exigidas para a aplicação dos métodos, na prática, são muito difíceis de serem satisfeitas, limitando o uso desses métodos.

3. Testes evolutivos

Nesta seção são apresentados o conceito de testes evolutivos e os trabalhos relacionados de teste funcional deste contexto.

3.1. Princípio

Recentemente, houve um grande interesse sobre teste baseado em busca (em inglês, *Search Based Testing*, SBT), que procura aplicar técnicas de otimização na geração de dados de teste. A idéia geral é que o conjunto das entradas possíveis forma um espaço de busca e o critério de teste é codificado como uma função objetivo. Essa abordagem natural torna SBT bastante atrativa para tratar diferentes problemas de geração de dados de teste, simplesmente mudando-se o espaço de busca e a função objetivo. Para cobertura de instruções, por exemplo, a função objetivo avaliaria quão próxima uma entrada de teste está de executar uma instrução não coberta.

Independente da técnica de otimização utilizada, apenas dois aspectos são necessários para aplicá-la em problemas de engenharia de software [Harman 2007]:

1. A escolha da representação do problema;
2. A definição da função objetivo.

Ênfase deve ser dada à função objetivo, pois captura as informações cruciais para a otimização. Através dos valores da função objetivo é que guia-se a busca, diferenciando uma boa solução de uma ruim [Harman 2007]. Além disso, uma boa escolha da representação do problema garante que soluções similares sejam “vizinhas” no espaço representacional, permitindo que a busca mova-se mais facilmente de uma solução a outra que compartilha um conjunto similar de propriedades [McMinn 2004].

Diferentes técnicas de otimização têm sido aplicadas no SBT, entre as mais utilizadas estão os algoritmos evolutivos. Essa aplicação de algoritmos evolutivos na geração

de dados de teste, referenciada normalmente na literatura como Testes Evolutivos, consiste em otimizar uma determinada entrada de acordo com o critério de teste expresso em uma função objetivo. Os indivíduos sobre os quais o processo de otimização ocorre constituem os casos de teste e estes são representados de maneira a permitir a aplicação de operadores genéticos. Os valores das funções objetivos são obtidos pela comparação das soluções da busca em relação ao seu objetivo geral, direcionando a busca para regiões promissoras do espaço de busca [McMinn 2004]. A seguir são apresentados alguns trabalhos relacionados aos testes evolutivos, no contexto de teste funcional.

3.2. Teste evolutivo: teste funcional

Algoritmos evolutivos têm sido aplicados na geração de dados de diferentes técnicas de teste, entre a mais investigada está o teste estrutural, mas também encontram-se esforços para o teste funcional. Nesse caso, os testes podem ser derivados de diferentes formas de especificação: notação Z [Jones et al. 1995], SPARK-Ada [Tracey et al. 1998] e MEF [Guo et al. 2004, Guo et al. 2005, Derderian et al. 2006].

Em relação aos trabalhos que derivam os teste a partir de MEFs, é investigada a construção de seqüência UIO (*Unique-Input-Output*) utilizando AG. As seqüências UIO são usadas no teste de MEF para verificar o estado final de uma seqüência de transição. Guo et al. [Guo et al. 2005] propõem uma função objetivo que guia a busca para potenciais seqüências UIO. Uma regra é definida para calcular o grau de similaridade entre os indivíduos, punindo aqueles que são muito similares a outros. Assim, esses indivíduos punidos têm menor probabilidade de serem selecionados para reprodução, ajudando a manter a diversidade da população. Essa abordagem busca resolver o problema de encontrar ótimos locais[1] apresentado em seu trabalho anterior [Guo et al. 2004].

Derderian et al. [Derderian et al. 2006] também descrevem um método para gerar UIOs de MEF utilizando AG. Porém, em relação ao trabalho de Guo et al. [Guo et al. 2004], há a diferença da máquina poder ser parcialmente especificada, uma vez que assumia-se uma máquina completamente especificada. Para tanto, as transições que faltam são ignoradas, mantendo a máquina no mesmo estado e a próxima entrada na seqüência é considerada. Nesse caso o valor de aptidão é penalizado. Além disso, a função objetivo é mais simples e utiliza uma classificação do número de ocorrência dos pares entrada/saída da máquina. O valor de aptidão é calculado como a soma das posições na classificação dos pares de entrada/saída que compõem a seqüência.

Neste trabalho, os casos de teste também são derivados a partir de uma MEF. Porém diferentemente dos trabalhos descritos acima nos quais a função objetivo busca encontrar seqüências UIO, a abordagem proposta neste artigo tem como finalidade cobrir um conjunto de transições determinadas pelo usuário. A abordagem é melhor descrita na Seção 5. Outra diferença é o algoritmo evolutivo adotado, sendo utilizado neste trabalho o algoritmo GEO, ao invés do AG. Na seção a seguir é apresentado o GEO.

4. GEO

Um algoritmo evolutivo proposto recentemente é o de Otimização Extrema Generalizada (em inglês, *Generalized Extremal Optimization*, GEO) [DeSousa 2002,

[1]Ótimos locais são pontos que minimizam/maximizam o valor de uma função, mas podem não ser o menor/maior valor possível da função.

DeSousa et al. 2003a]. Esse algoritmo foi desenvolvido como uma generalização do método da Otimização Extrema (em inglês, *Extremal Optimization*, EO) [Boettcher and Percus 2001], com o objetivo de tornar sua implementação independente do tipo de problema que está sendo atacado. Os processos internos tanto do EO quanto do GEO foram inspirados em um modelo simplificado de seleção natural para mostrar evidências da presença de Criticalidade Auto-Organizada (em inglês, *Self-Organized Criticality*, SOC) em ecossistemas naturais [Bak and Sneppen 1993].

A teoria do SOC propõe que sistemas complexos com muitos elementos que interagem entre si evoluem naturalmente para um estado crítico onde uma simples mudança em um de seus elementos produz perturbações que podem atingir qualquer número de elementos do sistema. A probabilidade de ocorrer uma perturbação de tamanho s é descrita por uma lei de potência na forma:

$$P(s) \sim s^{-\tau} \tag{1}$$

onde τ é um parâmetro positivo. Nota-se que perturbações de menor tamanho ocorrem mais freqüentemente que perturbações grandes, embora perturbações tão grandes quanto o sistema possam ocorrer com uma probabilidade não desprezível. Essa teoria tem sido utilizada para explicar o comportamento de sistemas complexos em áreas como geologia, economia e biologia [Bak 1996].

Bak e Sneppen [Bak and Sneppen 1993] desenvolveram um modelo simplificado de um ecossistema no qual as espécies são representadas lado a lado e possuem uma relação de vizinhança. A Figura 1 mostra n espécies, sendo que e_n é vizinho de e_{n-1} e e_1, e_1 é vizinho de e_n e e_2, e assim sucessivamente. Para cada espécie é associado um valor de aptidão no domínio $[0, 1]$. Aquela com pior valor de aptidão é considerada a espécie menos adaptada e o processo de evolução força esta e seus vizinhos a mudar. A mudança é feita atribuindo-se aleatoriamente novos valores de aptidão a essas espécies, podendo ocasionar a evolução ou extinção de qualquer uma delas, mesmo que os valores de aptidão não sejam melhores que os anteriores. Uma vez que a espécie menos adaptada é constantemente forçada a mudar, a aptidão média do ecossistema aumenta e, após algumas iterações, toda população apresenta um índice de adaptabilidade acima de um certo valor, denominado de valor crítico. Eventualmente os valores de aptidão de algumas espécies podem ficar abaixo do valor crítico por meio de avalanches (pertubações), cuja probabilidade de ocorrência também segue uma lei de potência na forma dada pela equação (1). Um método de otimização baseado nesse modelo poderia evoluir soluções rapidamente, sistematicamente modificando as espécies menos adaptadas e, ao mesmo tempo, poderia investigar regiões diferentes do espaço de projeto através das avalanches, escapando de mínimos locais.

Figura 1. População de espécies no modelo de Bak e Sneppen

Boettcher e Percus [Boettcher and Percus 2001] propuseram o método EO inspirado no modelo de Bak-Sneppen com o objetivo de tratar problemas difíceis de otimização

combinatória. O valor de aptidão é associado a cada variável de projeto, ao contrário do AG, no qual uma configuração de variáveis é associada a uma solução. Entretanto, os autores observaram que uma definição geral para o valor de aptidão para as variáveis individuais pode se mostrar ambígua ou mesmo impossível, sendo necessária uma nova definição do valor de aptidão para cada novo problema de otimização abordado pelo método. Isso motivou o desenvolvimento do algoritmo GEO para resolver esse problema e possibilitar sua aplicação a uma ampla categoria de problemas que incluem variáveis contínuas, discretas, inteiras ou uma combinação destas.

No GEO, cada bit é uma espécie e uma cadeia (*string*) binária é considerada uma população de espécies. A cadeia codifica as M variáveis de projeto e um valor de aptidão é associado a cada bit da cadeia. Esta é a diferença com relação ao EO, cujo valor de aptidão é associado a cada variável de projeto. A Figura 2 mostra as variáveis x e y codificadas em uma cadeia representada por quatro espécies no GEO e duas espécies no EO. Uma população de m bits que constituem uma cadeia representa uma solução para o problema.

Figura 2. Representação de uma população no GEO (4 espécies e 2 variáveis de projeto) e no EO (2 espécies e 2 variáveis de projeto)

Uma descrição resumida dos passos do GEO, adaptada de [DeSousa 2002], encontra-se a seguir:

1. Inicialize aleatoriamente a população de L espécies (bits) que codificam M variáveis de projeto.
2. Para cada bit atribua um índice de adaptabilidade que seja proporcional ao ganho ou perda que a função objetivo tem ao realizar uma mutação no valor do bit (de '0' para '1' ou de '1' para '0'), comparado com um dado valor de referência. O valor do bit retorna ao seu valor original.
3. Ordene os bits de acordo com seu índice de adaptabilidade, sendo o primeiro o menos adaptado. Se dois ou mais bits possuírem o mesmo índice de adaptabilidade, ordene-os aleatoriamente com distribuição uniforme.
4. Escolha com probabilidade uniforme um bit candidato i para sofrer mutação. Gere um número aleatório RAN com probabilidade uniforme no domínio [0,1]. Se a probabilidade de mutação para o bit i calculada por $P_i(k) = k^{-\tau}$, onde k é a posição de i na ordenação e τ um parâmetro ajustável e positivo, for igual ou menor que RAN, o bit i é modificado. Senão, escolha um novo bit candidato e repita o processo até que um bit seja modificado.
5. Repita os passos de 2 a 4 até que um critério de parada seja satisfeito.
6. Retorne a melhor configuração de bits (solução) encontrada durante a busca.

Em uma implementação prática do algoritmo descrito acima, a primeira decisão a ser tomada é definir o número de bits necessário para representar cada variável de projeto. Isso depende do domínio e precisão que se deseja para o valor de cada variável. Além

disso, também deve-se definir o valor do parâmetro τ antes de iniciar a busca. Esse é o único parâmetro ajustável do GEO, sendo que para cada problema existe um que torna a busca mais eficiente. Por exemplo, se $\tau \rightarrow \infty$, somente o primeiro bit da ordenação sofrerá mutação em cada iteração do algoritmo, o que é equivalente a uma busca totalmente determinística. Por outro lado, se $\tau \rightarrow 0$, qualquer bit escolhido como candidato (estando ou não em uma boa posição na ordenação) sofrerá mutação. A experiência tem mostrado que o melhor valor de τ para um dado problema varia normalmente entre 0.75 e 5.0. Ter apenas um único parâmetro livre a ser ajustado pode ser considerado *a priori* uma vantagem em comparação a outros algoritmos estocásticos, uma vez que estes normalmente possuem mais parâmetros livres, demandando mais esforço e tempo para serem ajustados.

5. Método proposto

O objetivo deste trabalho é gerar dados para o teste de conformidade, tendo MEF como especificação, através do algoritmo GEO. O critério de teste adotado é cobrir um determinado conjunto de transições, que podem ser não seqüenciais. O motivo pela escolha desse critério, ao invés de cobrir todas as transições, é que pretende-se gerar seqüências que cubram transições que sejam críticas ao sistema em teste. Por exemplo, pode-se testar as transições que correspondem às entradas inválidas para realizar o teste de robustez do sistema.

Na Figura 3 é apresentada uma ilustração do procedimento de geração dos dados para o teste de conformidade. É utilizada a ferramenta SMC[2] (*State Machine Compiler*) capaz de gerar um código que faz a simulação de uma máquina de estados nas linguagens Java, C++ e Perl, por exemplo. Como primeiro passo, uma MEF M é passada para a ferramenta SMC que, por sua vez, gera um código Java P que simula essa máquina. O código gerado é instrumentado para informar em quais transições uma seqüência de entradas passou. Em seguida, dado um conjunto de transições alvo T_{alvo}, o GEO começa a gerar dados de teste tentando cobrí-lo. Cada dado gerado é uma seqüência de entrada para M. O caminho coberto pela seqüência é obtido graças à instrumentação inserida em P e será entrada para a função objetivo que calculará a cobertura desse caminho em relação a T_{alvo}. O critério de parada do GEO é (i) quando se atinge um número máximo de avaliações da função objetivo ou (ii) quando todo conjunto T_{alvo} for coberto. Caso o critério de parada não seja satisfeito, o GEO reinicia o processo de geração de dados.

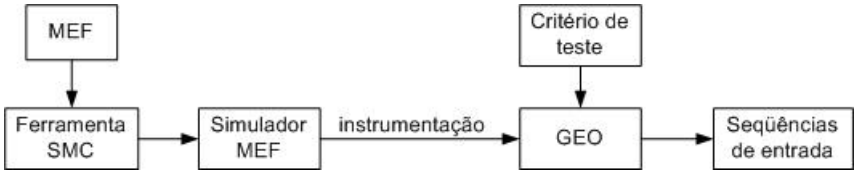


Figura 3. Ilustração da abordagem utilizada.

Antes de iniciar o processo para gerar dados de teste, deve-se definir em relação às variáveis de projeto: a quantidade, o tipo, o domínio que especifica o limite superior e inferior para seus valores e a precisão, como também deve-se definir o valor do único parâmetro ajustável do GEO τ e o número máximo de avaliações da função objetivo. É recomendável fazer um ajuste fino do parâmetro τ, visto que cada sistema pode ter uma ca-

[2]Disponível em http://smc.sourceforge.net.

racterística que faz com que valores diferentes deste parâmetro tenham grande influência na eficiência da geração dos dados de teste. Neste trabalho, cada variável de projeto representa um evento de entrada $x \in I$ da máquina e, consequentemente, a população é uma seqüência de eventos de entrada. O número de variáveis de projeto determina o tamanho da seqüência de teste gerada. O número de bits m necessário para representar cada variável de projeto depende do domínio das variáveis e da precisão p desejada, sendo dado pela inequação a seguir:

$$2^m \geq \left[\frac{max_j - min_j}{p} + 1 \right] \tag{2}$$

onde min_j e max_j são os limites inferior e superior da variável j, respectivamente. Caso o valor de m não seja um valor inteiro, então m passa a ser o próximo número inteiro imediatamente superior a ele. O valor real v_j da variável de projeto j é obtido pela equação abaixo:

$$v_j = min_j + (max_j - min_j) \cdot \left[\frac{Int_j}{2^m - 1} \right] \tag{3}$$

onde Int_j é o número inteiro obtido na transformação da variável j de sua representação binária para decimal.

5.1. Estudo de caso

Um estudo de caso da abordagem foi feito para o protocolo WTP (*Wireless Transaction Protocol*), uma das camadas do WAP (*Wireless Application Protocol*) cujo objetivo é oferecer acesso à Internet para dispositivos móveis, como telefone celular. WTP é um protocolo de transação confirmada que provê os serviços essenciais para aplicações interativas pedido/resposta. O protocolo permite balancear o grau de confiabilidade de um servidor de transação. O processo que inicia uma transação é denominado de *Initiator*, enquanto que o receptor é denominado de *Responder*.

As primitivas de serviço relativas à camada superior são: `TR-Invoke` (inicia uma nova transação), `TR-Result` (retorna um resultado de uma transação iniciada anteriormente) e `TR-Abort` (aborta uma transação existente). Os tipos de primitivas definidos são: (i) `req`, indica um pedido de um serviço da camada superior; (ii) `ind`, usada pela camada que provê o serviço para indicar à próxima camada superior as atividades relacionadas ao *peer* ou ao provedor do serviço; (iii) `res`, indica o recebimento do tipo de primitiva `ind` da próxima camada inferior; e (iv) `cnf`, informa que a atividade foi completada com sucesso.

Uma unidade de dado do protocolo (PDU) é composta por um cabeçalho e um dado (opcional). O cabeçalho contém uma parte fixa e uma variável. A parte fixa do cabeçalho contém o tipo de PDU e normalmente utiliza parâmetros. Os tipos básicos de PDU são: `invoke` (o *Initiator* inicia uma transação), `ack` (confirmação de uma mensagem recebida), `result` (o *Responder* responde a um `TR-Invoke` enviado pelo *Initiator* e `abort` (usado para abortar uma transação). Para maiores informações sobre o WTP, o protocolo [WAP Forum 2001] pode ser consultado.

A fim de testar o *Responder*, uma MEF M_1 foi construída considerando apenas suas operações. Embora exista um fluxo de dados no WTP, tais como variáveis e

parâmetros, somente o fluxo de controle foi levado em consideração até o momento na MEF. A máquina é determinística e parcialmente especificada, ignorando-se eventos de entrada que não foram especificados para um estado e mantendo-a no mesmo estado. A MEF M_1 possui 6 estados e 45 transições. Devido ao tamanho da máquina, esta não será apresentada neste artigo. O conjunto de transições a serem cobertas é aquele correspondente às exceções especificadas na documentação do protocolo. Por exemplo, pretende-se cobrir as transições correspondentes ao recebimento de um `abort` no *Responder*.

5.2. Função objetivo

Dado um conjunto de transições T_{alvo} a serem cobertas e uma seqüência de entrada seq da máquina, a função objetivo verifica as transições disparadas por seq e calcula o número de transições n comuns a T_{alvo}. O valor de aptidão de seq é definido como: $n/|T_{alvo}|$, em que $|T_{alvo}|$ é a cardinalidade do conjunto T_{alvo}. Assim os valores da função objetivo estão no domínio de [0,1]. Quando nenhuma transição de T_{alvo} é coberta por seq, o valor de aptidão é zero. Já seu valor máximo será 1 quando todas as transições de T_{alvo} forem cobertas por seq.

Deve-se ressaltar que outros critérios de adequabilidade poderiam ser utilizados, assim outras funções objetivo poderiam ser definidas para atender esses critérios.

5.3. Resultados

Como o desempenho do GEO varia significativamente com o valor do parâmetro ajustável τ, uma série de experimentos foi realizada para configurá-lo adequadamente ao problema do teste de conformidade. Foram realizados 10 experimentos para diferentes números de variáveis de projeto, que corresponde ao tamanho da seqüência de teste gerada, variando entre 4 e 24, com incremento de 4. Cada experimento é equivalente a 50 execuções com uma configuração de parâmetro. Para cada execução, o número máximo de avaliações da função objetivo é 25000. O valor do parâmetro τ inicia-se com zero e é incrementado de 0.25 em cada experimento até atingir valor 5, então $0.75 \leq \tau \leq 5$. O gráfico da Figura 4 mostra a influência de τ no desempenho do GEO para obter execuções com sucesso ao encontrar todas as transições de T_{alvo}. Pode-se observar que, com o valor de τ igual a 0.75, a maioria das execuções cobrem T_{alvo} independente do tamanho da seqüência. Também nota-se que conforme o número de variáveis de projeto aumenta, melhor é a cobertura de T_{alvo}, o que já era de se esperar, uma vez que há mais possibilidade da seqüência possuir transições de T_{alvo}.

Todos os experimentos foram realizados usando-se o tipo inteiro e apenas os inteiros positivos foram considerados. Em M_1, existem 23 eventos de entrada, assim as variáveis de projeto estão no intervalo [0, 22]. Com essas informações, o número de bits para representar cada variável de projeto é calculado pela relação (2), com precisão p igual a 1. As transições pertencentes a T_{alvo} são aquelas em que o evento de entrada equivale a uma exceção especificada na documentação do protocolo WTP e constitui um total de 25 transições em M_1. Considerando 4 variáveis de projeto, um dado de teste gerado é uma seqüência de 4 eventos de entrada, por exemplo: (`ack`, `invoke`, `TR-Result.req`, `abort`). Nesse caso, como `ack` não é esperado no estado inicial s_0, a máquina ignora esse evento e permanece em s_0. Em seguida, os demais eventos são disparados, visto que existe esse caminho na máquina. A transição correspondente ao evento `abort` está contida em T_{alvo}, assim o valor da função objetivo para essa seqüência é igual a $1/25 = 0,04$.

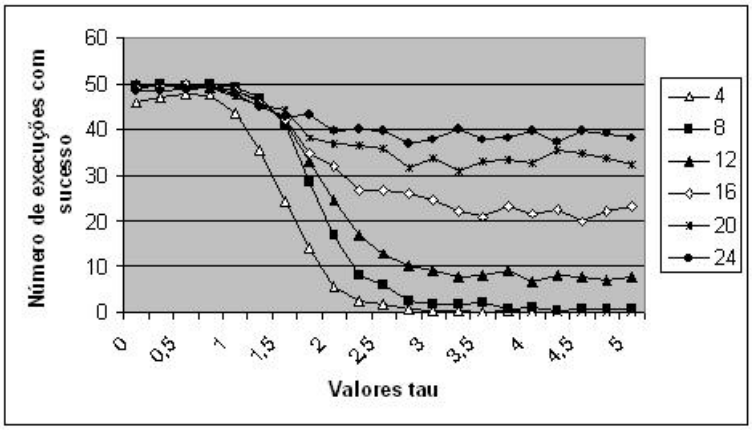


Figura 4. Configuração do parâmetro τ.

A fim de avaliar a aplicação da abordagem, o GEO foi comparado com o teste aleatório. O teste aleatório realiza uma busca na qual dados de teste são selecionados arbitrariamente do domínio de entrada para serem aplicados ao sistema em teste. A geração de dados de teste é feita atribuindo-se valores de '0' ou '1' aleatoriamente para cada bit necessário para representar um candidato a dado de teste. Em seguida, esse candidato é convertido para um valor inteiro e está pronto para ser aplicado no sistema em teste. A comparação do algoritmo GEO e o teste aleatório foi realizada com base na porcentagem média de cobertura em função dos dados de teste gerados. Um experimento de 10 execuções com máximo de 50000 avaliações da função objetivo foi feito para diferentes números de variáveis de projeto, iniciando-se de 4 até 20, com incremento de 4. No caso do GEO, o experimento foi executado com o melhor valor de τ ($\tau = 0.75$). O resultado dessa comparação com 8 variáveis de projeto é mostrado na Figura 5. Verifica-se que o teste aleatório alcançou rapidamente a cobertura total, devido principalmente ao fato do domínio de geração ser pequeno, que são valores do intervalo [0,22]. Embora o GEO necessite um número maior de execuções, o algoritmo também conseguiu obter um conjunto de casos de teste que cubra T_{alvo}. Lembrando que o algoritmo do teste aleatório é bastante simples e, possivelmente em um exemplo mais complexo, seu desempenho não deve se manter.

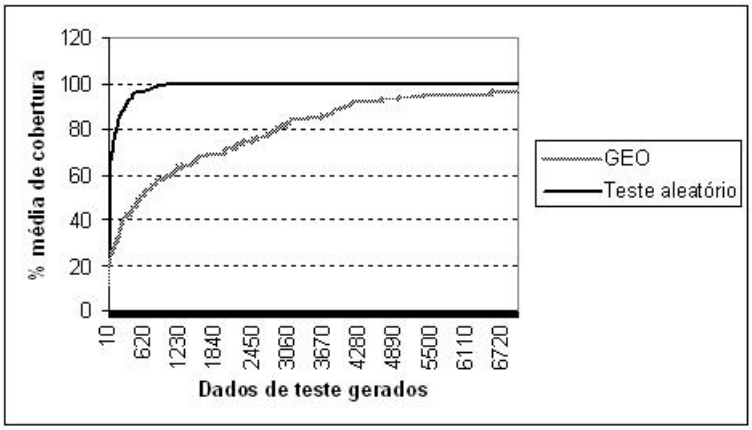


Figura 5. GEO X Teste aleatório.

A evolução dos melhores valores de aptidão no GEO com números de variáveis diferentes é mostrada na Figura 6. Observa-se que após uma média de 7000 avaliações da função objetivo, os valores de aptidão não variam muito.

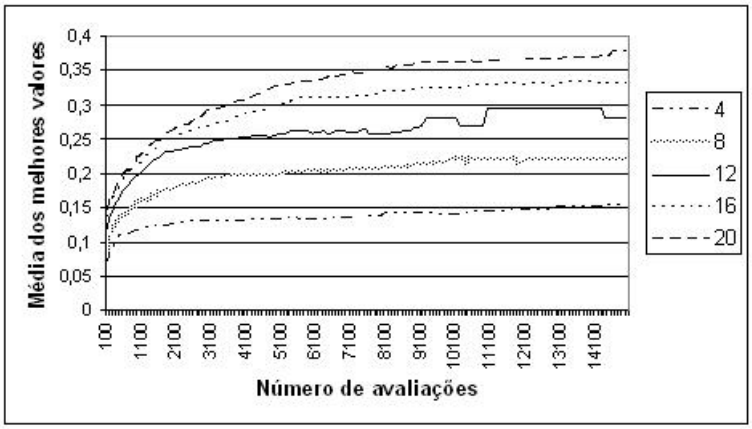


Figura 6. Evolução dos melhores valores da função objetivo no GEO.

6. Conclusões e trabalhos futuros

No contexto de testes evolutivos, o teste funcional é ainda pouco explorado. Assim neste trabalho busca-se aplicar um algoritmo evolutivo no teste de conformidade, possuindo MEF como especificação. Diferentemente da maioria dos trabalhos que aplicam o AG, foi utilizado o algoritmo GEO cujo processo de configuração é mais simples que o AG.

Visto que a função objetivo é bastante simples, deve-se investigar outras funções objetivos que utilizam o conhecimento da estrutura do modelo para melhorar o processo de busca por melhores soluções.

Este trabalho é o passo inicial para a investigação do algoritmo GEO no teste baseado em modelo. Pretende-se utilizar MEF estendida (MEFE), que permite representar o fluxo de dados de um sistema, ao invés do modelo clássico que apenas representa o fluxo de controle. Um problema a ser tratado nas MEFEs é que podem existir seqüências não executáveis. A questão de quais seqüências de entradas devem ser aplicadas para permitir que a implementação em teste execute uma determinada transição é indecidível, ou seja, é impossível encontrar um algoritmo que retorne tal seqüência de entrada ou retorne a mensagem como “esta transição não é executável” [Dssouli et al. 1999]. Assim, métodos de teste de conformidade que utilizam especificações baseadas em MEFE necessitam de heurísticas para tornar as seqüências executáveis. Essa é a razão pela qual foi utilizado um algoritmo evolutivo no teste baseado em modelo e não os métodos já tradicionalmente conhecidos na literatura.

Referências

Abreu, B. T. (2006). Uma abordagem evolutiva para a geração automática de dados de teste. Master’s thesis, IC/Unicamp, Campinas, SP.

Bak, P. (1996). *How nature works: the science of self-organized criticality*. Springer-Verlag, New York.

Bak, P. and Sneppen, K. (1993). Punctuated equilibrium and criticality in a simple model of evolution. *Physical Review Letters*, 71(24):4083–4086.

Bertolino, A. (2007). Software testing research: Achievements, challenges, dreams. In *FOSE '07: 2007 Future of Software Engineering*, pages 85–103, Washington, DC, USA. IEEE Computer Society.

Bochmann, G. and Petrenko, A. (1994). Protocol testing: Review of methods and relevance for software testing. In *International Symposium on Software Testing and Analysis (ISSTA'94)*, pages 109–124.

Boettcher, S. and Percus, A. G. (2001). Optimization with extremal dynamics. *Physical Review Letters*, 86:5211–5214.

Bourhfir, C., Dssouli, R., and Aboulhamid, E. M. (1996). Automatic test generation for EFSM-based systems. Technical Report IRO 1043, University of Montreal, Canada.

Davis, A. M. (1988). A comparison of techniques for the specification of external system behavior. *Communications of the ACM*, 31(9).

Derderian, K., Hierons, R., Harman, M., and Guo, Q. (2006). Automated Unique Input Output sequence generation for conformance testing of FSMs. *The computer Journal*, 49(3):331–344.

DeSousa, F. L. (2002). *Otimização Extrema Generalizada: Um novo algoritmo estocástico para o projeto ótimo*. PhD thesis, INPE, São José dos Campos, SP, Brasil.

DeSousa, F. L., Ramos, F. M., Paglione, P., and Girardi, R. M. (2003a). New stochastic algorithm for design optimization. *AIAA Journal*, 41(9):1808–1818.

DeSousa, F. L., Vlassov, V., and Ramos, F. M. (2003b). Generalized extremal optimization for solving complex optimal design problems. In *GECCO '03: Proceedings of the 2003 Genetic and Evolutionary Computation Conference*, volume 2723 of *Lecture Notes in Computer Science*, pages 375–376.

DeSousa, F. L., Vlassov, V., and Ramos, F. M. (2004). Generalized extremal optimization: An application in heat pipe design. *Applied Mathematical Modelling*, 28(10):911–931.

Dssouli, R., Saleh, K., Aboulhamid, E., En-Nouaary, A., and Bourhfir, C. (1999). Test development for communication protocols: towards automation. *Computer Networks*, 31(17):1835–1872.

Fujiwara, S., Bochmann, G., Khendek, F., Amalou, M., and Ghedamsi, A. (1991). Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603.

Gallaher, M. P. and Kropp, B. M. (2002). Economic impacts of inadequate infrastructure for software testing. Technical Report RTI Project Number 7007.011 - NIST Planning Report 02-3, University of Montreal.

Galski, R. L., de Sousa, F. L., Ramos, F. M., and Muraoka, I. (2007). Spacecraft thermal design with the generalized extremal optimization algorithm. *Inverse Problems in Science and Engineering*, 15(1):61–75.

Geurts, W., Wijbrans, K., and Tretmans, J. (1998). Testing and formal methods - BOS project case study. In 6^{th} *European International Conference on Software Testing, Analy-*

sis & (ReviewEuroSTAR'98), pages 215–229, Munich, Germany. Aimware, Mervue, Galway, Ireland.

Gill, A. (1962). *Introduction to the Theory of Finite-State Machines*. McGraw-Hill, New York.

Guo, Q., Hierons, R. M., Harman, M., and Derderian, K. (2004). Computing unique input/output sequences using genetic algorithms. In Petrenko, A. and Ulrich, A., editors, *3rd International Workshop on Formal Approaches to Testing of Software (FATES 2003)*, volume 2931 of *Lecture Notes in Computer Science*, pages 164–177. Springer.

Guo, Q., Hierons, R. M., Harman, M., and Derderian, K. (2005). Constructing multiple unique input/output sequences using metaheuristic optimisation techniques. *IEE Proceedings — Software*, 152(3):127–140.

Harman, M. (2007). The current state and future of search based software engineering. In *Future of Software Engineering 2007*. IEEE Computer Society.

Harman, M. and McMinn, P. (2007). A theoretical & empirical znalysis of evolutionary testing and hill climbing for structural test data generation. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 73–83, New York, NY, USA. ACM.

Harrold, M. J. (2000). Testing: a roadmap. In *ICSE - Future of SE Track*, pages 61–72.

Jones, B., Sthamer, H., Yang, X., and Eyres, D. (1995). The automatic generation of software test data sets using adaptive search techniques. In *3rd International Conference on Software Quality Management*, pages 435–444, Seville, Spain.

Korel, B. (1990). Automated software test data generation. *IEEE Trans. Softw. Eng.*, 16(8):870–879.

McMinn, P. (2004). Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2):105–156.

Michael, C., McGraw, G., and Schatz, M. A. (2001). Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110.

Offutt, A. J. (1991). An integrated automatic test data generation system. *Journal of Systems Integration*, 1(3-4):391–409.

Pargas, R. P., Harrold, M. J., and Peck, R. R. (1999). Test-data generation using genetic algorithms. *The Journal of Software Testing, Verification and Reliability*, 9(4):263–282.

Tracey, N., Clark, J., and Mander, K. (1998). Automated program flaw finding using simulated annealing. In *ISSTA '98: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, pages 73–81. ACM Press.

WAP Forum (2001). WAP wireless transaction protocol specification, wap-224-wtp-20010710-a.

Watkins, A. and Hufnagel, E. M. (2006). Evolutionary test data generation: a comparison of fitness functions: Research articles. *Software Practice and Experience*, 36(1):95–116.

Teste por injeção de falhas da implementação do protocolo de comunicação SCTP

Sergio Cechin[1], Werner Nedel[1], Taisy Weber[1]

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil
{cechin,wmnedel,taisy}@inf.ufrgs.br

Abstract*: The advantages of using a fault injector to test the implementation of communication protocols are discussed. SCTP is a new protocol over IP that offers services to network applications that are similar to UDP and TCP. SCTP promises to these applications high reliability guarantees. Injecting communication faults into SCTP, we show the feasibility of the evaluation of protocol behavior under faults. FIRMAMENT is the injector chosen for the test experiments because its availability in our laboratories, but mainly on account of its ability to operate directly along the kernel protocol stack and the easiness to be configured to handle several communication protocols.*

Resumo*. São apresentadas as vantagens do uso de um injetor de falhas de comunicação para o teste da implementação de protocolos de rede. SCTP é um protocolo novo, que oferece serviços semelhantes aos do UDP e do TCP e promete garantir confiabilidade às aplicações de rede. Através de experimentos de injeção de falhas sobre o protocolo SCTP mostra-se como é possível avaliar o comportamento do protocolo na presença de falhas controladas. O injetor FIRMAMENT foi escolhido para os experimentos de teste por estar disponível para uso, por possuir um alto poder de expressão de carga de falhas, por operar diretamente na pilha de protocolos do sistema operacional e pela facilidade com que pode ser configurado para operar com diferentes protocolos de rede.*

1 Teste sob falhas da implementação de protocolos de comunicação

Qualquer artefato de software que deva atender requisitos mínimos de confiabilidade e disponibilidade deve ser desenvolvido prevendo a ocorrência de falhas e o tratamento ou sinalização dos erros por elas ocasionados. Os procedimentos implementados para a detecção e tratamento de erros devem ser testados em um ambiente sujeito a falhas. Para não depender da ocorrência natural de falhas, cuja freqüência pode ser muito baixa e a controlabilidade nula, uma técnica adequada é emular falhas por software e aplicá-las à implementação sob teste.

Protocolos de comunicação são desenvolvidos em software seguindo uma dada especificação. Por atuarem nos níveis mais baixos de abstração, uma implementação incorreta destes protocolos apresenta um enorme potencial de interferir negativamente no comportamento dos níveis superiores, prejudicando as aplicações que dela dependam e, portanto, qualquer serviço de rede fornecido por software.

A execução de um protocolo de comunicação é tipicamente determinada pelo estado de cada participante e pelas mensagens que os participantes trocam entre si. É muitas vezes impossível ou inconveniente alterar diretamente o estado do protocolo, mas para levar o protocolo a um determinado comportamento observável é possível atuar diretamente sobre as mensagens. Por exemplo, em um protocolo especificado para tolerar perdas de mensagens ou colapso de um fluxo, é possível testar seu comportamento a esses tipos de falha simplesmente suprimindo algumas ou todas as mensagens que um participante recebe.

Uma ferramenta atuando sob o protocolo alvo na pilha de protocolos é extremamente útil para o teste. Essa ferramenta pode observar e atuar sobre qualquer mensagem enviada ou recebida. Desta forma o desenvolvedor ou testador do protocolo pode suprimir, alterar valores de campos e atrasar cada uma das mensagens recebidas ou enviadas, controlando o momento preciso da ocorrência de uma falha, e assim verificar o comportamento da sua implementação em situações de maior risco. Para o testador ou certificador da implementação, uma ferramenta na pilha de protocolos permite também emular parâmetros como freqüência de omissão de mensagens e atrasos crescentes por sobrecarga de rede, úteis para obter medidas estatísticas sobre o comportamento do protocolo sob determinada taxa de falhas de comunicação.

SCTP é um protocolo relativamente novo, descrito inicialmente na RFC2960 [Stewart 2000] e consolidado na RFC4960 [Stewart 2006], originalmente desenvolvido para transmitir mensagens de sinalização da Rede de Telefonia Pública Comutada sobre IP. SCTP fornece associações entre clientes e servidores, podendo oferecer múltiplos fluxos entre pontos finais de conexão, cada um com sua própria entrega confiável de mensagens. O protocolo SCTP promete conferir alta confiabilidade às aplicações de rede.

Neste artigo relatamos um experimento de injeção de falhas sobre o SCTP conduzido com o objetivo de ilustrar as vantagens do uso de injetores de falhas no nível da pilha de protocolos para o teste da implementação de protocolos de comunicação. É mostrado como é possível avaliar o comportamento do SCTP em presença de falhas controladas. Entre os injetores de falhas mencionados na literatura para experimentação com protocolos de comunicação, FIRMAMENT foi escolhido principalmente por ter sido desenvolvido e estar disponível nos laboratórios onde o trabalho foi conduzido, mas também por possuir um alto poder de expressão de carga de falhas, por operar diretamente na pilha de protocolos do sistema operacional e por permitir ser facilmente configurado para trabalhar com diferentes protocolos de rede.

Nas próximas seções serão apresentados os conceitos básicos de condução de testes de injeção de falhas, a ferramenta usada, o protocolo alvo SCTP e os experimentos conduzidos sobre o SCTP.

2 Injeção de falhas

Um problema é saber se a estratégia empregada para detectar falhas e corrigir os erros provocados por essas falhas resulta realmente em aumento de confiabilidade. Como na maior parte das redes de comunicação as taxas de falhas são relativamente baixas e as falhas acontecem de forma aleatória e incontrolável, o problema consiste em avaliar se a estratégia empregada está realmente tolerando as falhas para as quais foi planejada, sem

necessidade de esperar indefinidamente para que as falhas significativas para uma dada situação realmente ocorram durante o teste. Uma solução é a injeção de falhas.

A injeção de falhas é um procedimento de teste das estratégias de tolerância a falhas empregadas. Através da introdução controlada de falhas pode se determinar se a estratégia permite ao sistema tolerar as falhas injetadas e qual o custo em desempenho relacionado à cobertura de falhas alcançada. Na fundamentação teórica, essa abordagem é explorada há tempo suficiente para ser considerada madura ([Arlat 1990], [Hsueh 1997]). Injetores de falhas têm sido construídos e aplicados ([Han 1995], [Carreira 1998], [Chandra 2004], [Locker and Xu 2003], [Martins 2002], [Some 2001], [Stoot 2000]). Raras, entretanto, são as ferramentas realmente disponíveis e, mesmo essas, são específicas para alguns domínios restritos de aplicações. Entre elas, mais raras ainda são os injetores de falhas que permitam emular falhas de rede e assim testar protocolos de comunicação. A complexidade dos sistemas de conexão e comunicação facilita a propagação de falhas e dificulta sua detecção. Sem controle sobre o tipo e origem das falhas torna-se extremamente difícil validar a correção das estratégias de tolerância a falhas nesses sistemas.

Apesar de madura na teoria, a experiência acumulada com teste sob falhas de sistemas baseados em troca de mensagens ainda não é suficiente. A experimentação com alvos reais, como implementações de protocolos de rede, traz como benefícios uma maior familiaridade com a área de testes por injeção de falhas e a possibilidade de avaliar também a própria ferramenta usada visando sugestões para a continuidade de seu desenvolvimento.

3 O protocolo SCTP

O SCTP – *Stream Control Transmission Protocol* – é um protocolo de nível de transporte, confiável, capaz de operar sobre um serviço de pacotes não confiável e sem conexão, como é o caso do IP. A confiabilidade oferecida por este protocolo é alcançada através do uso de mensagens de confirmação (*acknowledge messages*) com eventual retransmissão, em caso de erro na recepção das mensagens. A detecção de erros nas mensagens é obtida com o uso de CRC e a duplicação de mensagens é evitada através de uma numeração seqüencial das mesmas.

O protocolo SCTP está definido na RFC4960 [Stewart 2006], que torna obsoletos os dois documentos de definição anteriores: a RFC2960 [Stewart 2000] e a RFC3309 [Stewart 2004]. Esse protocolo foi originalmente projetado para a transmissão, sobre IP, de mensagens de sinalização da RTPC – Rede de Telefonia Pública Comutada. Entretanto, devido às suas características, outros tipos de aplicação podem obter vantagens em utilizá-lo. De uma forma geral, o protocolo SCTP fornece os seguintes serviços:

- transferência de dados sem erro e sem duplicação de mensagens;
- fragmentação de pacotes para ajustar ao MTU da rota selecionada;
- entrega seqüencial de mensagens, com possibilidade de ordenação por usuário, mesmo que transmitidas em múltiplos fluxos;
- possibilidade de empacotamento de múltiplas mensagens de usuário em um único pacote SCTP;

- suporte a tolerância a falhas pelo uso de múltiplos encaminhamentos para o destino.

3.1 Aplicação do SCTP

Apesar do TCP ser o principal protocolo para a transmissão confiável de dados sobre redes IP, um número crescente de aplicações têm sido desenvolvidas sobre UDP, de maneira a utilizarem mecanismos próprios para prover confiabilidade de transmissão. Dois são os fatores que levam a essa decisão de projeto: a inflexibilidade com que o protocolo TCP efetua o controle de erros, não levando em consideração as peculiaridades das aplicações, e o (baixo) desempenho apresentado, quando se compara uma conexão TCP com uma solução especificamente projetada para a aplicação (usando UDP, por exemplo).

O TCP oferece a transferência confiável e a ordenação estrita na entrega de pacotes. Entretanto, as aplicações podem necessitar a transferência confiável, porém sem a ordenação. Outras, ainda, podem requerer um ordenamento parcial das mensagens sem a necessidade da transferência confiável. Note-se que essas aplicações, se utilizassem o TCP, apresentariam um baixo desempenho relativo, causado por um mecanismo que não lhes é útil. Dessa forma, o SCTP possibilita que a aplicação configure, de forma independente, o uso da confirmação de entrega de mensagens e ordenamento das mesmas, permitindo a escolha da operação de forma semelhante ao do UDP (sem confirmação e sem ordenamento) ou, no outro extremo, com ambas: confirmação e ordenamento total.

Diferentemente do protocolo TCP, o protocolo SCTP é orientado a mensagens ao invés de um fluxo contínuo de bytes. Dessa forma, cada pacote é recebido atomicamente, como um bloco indivisível, exatamente da forma que foi transmitido. Devido a essa característica, o serviço de entrega de pacotes do SCTP pode ser descrito como um serviço de datagramas (pois são entregues na forma como foram transmitidos) com confirmação (devido ao mecanismo de confirmação de entrega das mensagens). Apesar de o SCTP oferecer um serviço de transmissão orientado a mensagens, as aplicações não necessitam fragmentar seus dados de acordo com um tamanho máximo de pacote. O próprio protocolo SCTP encarrega-se da fragmentação e remontagem dos fragmentos no destino.

Note-se que, no caso do TCP, a aplicação não tem a informação de início ou término das mensagens, cabendo às aplicações incorporarem delimitadores de registros aos dados transmitidos, de maneira a possibilitar a sua separação no destino. Além disso, no caso de uso do TCP e quando a aplicação encerra o envio dos dados, esta deve limpar o buffer de transmissão de maneira que todos os bytes sejam transmitidos. Isso não é necessário com o SCTP, uma vez que o protocolo incorpora um mecanismo que permite o envio e a recepção dos pacotes, mesmo após o encerramento da "associação" (forma mais geral de conexão, usada pelo SCTP).

Ainda, o TCP é capaz de associar um único fluxo de dados a uma conexão. Assim, a transmissão de múltiplos fluxos deve ser feita através de múltiplas conexões. De forma diferente, um pacote SCTP é capaz de transportar vários fluxos, uma vez que oferece serviços separados para cada um deles.

3.2 Associações SCTP

Os pacotes SCTP, diferentemente do TCP, não são transmitidos em uma conexão, mas em uma associação. Uma conexão utiliza apenas dois terminais e um caminho entre eles enquanto que uma associação é mais abrangente, pois possibilita que vários fluxos de dados sejam negociados entre os terminais.

Uma associação também permite que os terminais de origem e destino tenham associados vários endereços de transporte (combinação de endereço de rede, tipicamente IP, e de uma porta SCTP), possibilitando que um fluxo possa ser transportado através de várias rotas, o que oferece redundância de encaminhamento como forma de tolerância a falhas. Além disso, cada fluxo de uma associação possui um controle de erros e de ordenação de entrega de mensagens independentes, evitando a interferência entre fluxos. Essa característica é especialmente interessante quando ocorre a perda ou o atraso de mensagens em algum dos fluxos. Um exemplo de interferência, que é solucionado por esse mecanismo, é o bloqueio de transmissão conhecido como *head-of-line*, ao qual o TCP está sujeito.

3.3 Pacote SCTP

As PDU – Protocol Data Unit –, ou pacotes SCTP, são transportados como *payload*s dos datagramas IP. Estes, por sua vez, para permitirem alcançar as funcionalidades citadas anteriormente, devem apresentar uma estrutura hierárquica que permita, por exemplo, encapsular mais de um fluxo de dados em um único pacote.

O pacote SCTP corresponde ao primeiro nível da hierarquia de estruturas de dados e o cabeçalho comum desses pacotes contém informações relativas ao pacote como um todo. Fazem parte desse cabeçalho as portas SCTP de origem e destino, uma identificação da associação (*tag*) e dígitos de verificação para controle de erros de transmissão.

A área de dados dos pacotes SCTP é dividida em *chunks*, cada um deles correspondendo a um fluxo de dados: esses *chunks* formam o segundo nível da hierarquia e são compostos, também, de um cabeçalho e uma área de dados. No cabeçalho estão identificados o tipo de dados contidos no *chunk*, um conjunto de flags de controle e o tamanho da área de dados do *chunk*.

A estrutura da área de dados de cada *chunk* depende de seu tipo. Os *chunks* do tipo DATA (que são usados para transportar dados) carregam a informação do TSN – *Transmission Sequence Number* –, usado para determinar eventuais falhas na recepção de *chunks*, a identificação do fluxo, usada para separar os vários fluxos de dados de uma determinada associação, e o SSN – *Stream Sequence Number* –, usado para ordenar os dados recebidos em um determinado fluxo.

Existem 15 tipos pré-definidos de *chunks* que são usados de maneira a obter a comunicação confiável e com ordenamento de entrega de mensagens. Assim, além dos *chunks* usados para a transferência de dados, existem *chunks* para inicializar e encerrar uma associação, para verificar a conectividade entre transmissor e receptor (mecanismo de *heartbeat*), para informar que uma mensagem foi recebida com erro (*acknowledge*), para sinalização de erros, etc.

3.4 Tolerância a falhas com SCTP

O formato do pacote SCTP permite que vários fluxos sejam encapsulados em um mesmo pacote. Dessa forma, pode-se manter várias conexões entre dois nodos quaisquer, sem que haja interferência entre eles. Além disso, a informação de uma mensagem foi entregue corretamente é enviada na forma de um *chunk* especial de reconhecimento. Além disso, uma vez que cada fluxo possui identificação própria e uma seqüência de numeração independente, a confiabilidade de entrega assim como a ordenação são, efetivamente, realizados no nível de fluxo de informação e não no nível de pacote (tal como acontece no TCP).

O SCTP é capaz de fragmentar e remontar, automaticamente, pacotes maiores do que o MTU da rota utilizada. Também é capaz de receber pacotes por várias rotas e processá-los juntos, discriminando os vários fluxos existentes nos pacotes recebidos. Essas duas funcionalidades são possíveis graças à numeração seqüencial existente no *chunks*. Dessa forma, os dados de um fluxo podem ser enviados através de várias rotas e, ao chegarem ao destino, serão corretamente identificados e remontados para entrega à aplicação. Essa funcionalidade possibilita a diversidade de rotas, o que torna o protocolo robusto diante de falhas de comunicação.

O controle de erros através de mensagens de reconhecimento está sempre ativo no SCTP. Entretanto, a ordenação dos *chunks* pode ser ativada ou desativada, sendo que, quando ativada, a ordenação é parcial, ou seja, os dados são ordenados dentro de cada fluxo de dados. O receptor é informado se deve aplicar o algoritmo de ordenação ou não através de um flag no conjunto de flags do cabeçalho do *chunk*.

4 O ambiente de teste

O ambiente de teste emula situações reais de funcionamento do sistema alvo sobre uma rede Internet injetando falhas através de um injetor e observando o comportamento da aplicação. Esse teste experimental permite determinar medidas como cobertura de falhas e queda de desempenho em caso de falhas, entre outras.

O ambiente usado no teste sob falhas do SCTP é composto pelo injetor de falhas, uma carga de trabalho e uma carga de falhas a ser injetada. A carga de falhas é construída de acordo com o modelo de falhas relacionado ao protocolo alvo e previsto na sua especificação. Cada experimento é um teste executado em condições controladas no qual são introduzidas as falhas descritas na carga de falhas. Os dados coletados durante um experimento são posteriormente analisados e as medidas desejadas calculadas.

A injeção de falhas em um protocolo de comunicação consiste basicamente em interpretar o conteúdo da mensagem, analisar a carga de falhas, selecionar as mensagens de interesse e atuar de alguma maneira sobre a mensagem [Dawson 96]. Essa atuação pode ocorrer de diversas maneiras, como duplicando ou descartando mensagens e/ou modificando do seu conteúdo.

4.1 O injetor de falhas

FIRMAMENT (*Fault Injection Relocatable Module for Advanced Manipulation and Evaluation of Network Transports*) [Drebes 2005], desenvolvido nos laboratórios do Grupo de Tolerância a Falhas da UFRGS, é um injetor localizado no kernel do sistema

operacional Linux, que permite injeção por software de falhas de comunicação no protocolo IP ou qualquer outro protocolo construído sobre IP. FIRMAMENT é a evolução de outro injetor do Grupo, ComFIRM [Drebes 2006], provendo um melhor mecanismo para a descrição de cargas de falhas. A funcionalidade desses injetores foi influenciada por ORCHESTRA [Dawson 96]. Os tipos de falhas injetáveis por FIRMAMENT são: omissão de envio e recebimento de mensagens, colapso de nodo e canal, atraso de mensagens e falhas bizantinas. Duas idéias básicas norteiam o injetor: possibilitar a modificação da configuração e da descrição dos cenários de falhas de um experimento durante sua execução e representar cargas de falhas através de regras simples, que não sobrecarreguem o processamento de mensagens.

As primeiras versões de ComFIRM exigiam a alteração do código do kernel do sistema operacional hospedeiro. Apesar de eficiente, essa estratégia é intrusiva no kernel e compromete a portabilidade do injetor. A sua última versão já adotava o conceito de módulo carregável no kernel. Esse conceito foi aproveitado em FIRMAMENT.

O injetor associa ao processamento dos pacotes de comunicação a execução de *scripts* de processamento eficiente, chamados *faultlets*, responsáveis por selecionar e atuar sobre as mensagens. FIRMAMENT provê portabilidade via módulo do kernel e através do uso de ganchos da interface NetFilter [Russel and Welte 2002] para acessar o fluxo de execução dos protocolos IPv4 e IPv6. NetFilter está disponível a partir da versão 2.4 do Linux. Assim, qualquer versão do kernel do Linux posterior suporta o injetor, sem a necessidade de recompilação do kernel.

4.2 Programação de cargas de falhas

Faultlets, definidos por FIRMAMENT, permitem um alto poder de expressão na descrição de cargas de falhas. Para usar o injetor, o testador programa um *faultlet* específico para cada experimento. Um *faultlet* vai ser tanto mais difícil de ser programado quanto mais complexo for o protocolo sendo testado e a cobertura desejada para o teste. Alguns injetores de falhas como FIONA [Jacques-Silva 2006] e ComFIRM possuem uma séria limitação na descrição de cargas de falhas, pois testes de valores em determinados campos da mensagem são feitos a partir de deslocamentos fixos na mensagem. Esse empecilho dificulta testes como nos cabeçalhos do protocolo IPV4, que possuem tamanho variável. FIRMAMENT não apresenta esse inconveniente.

Um *faultlet* é executado sobre cada pacote que cruza um dos fluxos de comunicação, podendo atrasá-lo, alterar seu conteúdo, duplicá-lo, descartá-lo ou aceitá-lo. Além das ações sobre pacotes, um *faultlet* opera sobre variáveis de estado do fluxo, que podem ser usadas para alterar o valor dos dados do pacote. Pode-se também executar operações lógicas e aritméticas, gerando uma maior flexibilidade de descrição de cenários de falhas para qualquer protocolo.

Um *faultlet* é interpretado por uma máquina virtual no injetor, que associa o pacote ao *faultlet* e às variáveis de estado. Essa máquina virtual opera sobre IPV4 e IPV6, possibilitando a descrição de cargas de falhas para quatro fluxos de dados. As variáveis de estado da máquina virtual são um conjunto de 16 registradores de uso geral de 32 bits, disponíveis para cada fluxo. Os registradores operam com representação de números inteiros com sinal no formato de rede, sendo que o injetor faz a sua conversão automática para o formato nativo do hardware onde está sendo executado. São 31 as

instruções disponíveis divididas em sete classes: entrada e saída, lógicas e aritméticas, ação sobre o pacote, desvio, manipulação de auto-incremento, manipulação de seqüência de caracteres e propósitos gerais. O injetor possui também uma opção de "cão-de-guarda" para evitar que um *faultlet* entre em laço eterno.

FIRMAMENT não interpreta diretamente a forma textual de um *faultlet*. A ferramenta dispõe de um montador, que verifica o tipo dos parâmetros das instruções e, se estiverem corretos, gera uma saída binária. A verificação ajuda a evitar o carregamento de *faultlets* mal descritos no injetor.

4.3 Arquivos de controle e configuração do injetor

Por ser um módulo do kernel, o injetor FIRMAMENT não dispõe de uma interface de alto nível para sua operação. A ferramenta é controlada através da interface de arquivos virtuais do sistema operacional, no diretório /proc/net/firmament. Encontram-se, neste diretório, os arquivos de regras, que permitem a leitura e escrita dos *faultlets* já em formato binário, para cada fluxo de pacotes, e o arquivo de controle, para o qual são passados os comandos de controle do injetor. Esses comandos permitem iniciar e parar o processamento do *faultlet*, mostrar registradores da máquina virtual do injetor e reiniciar a ferramenta, parando todos os fluxos e eliminando todos os *faultlets* configurados.

Para análise dos resultados de um experimento de injeção de falhas é necessário registrar os eventos realizados pelo injetor, o que permite relacionar os resultados com as alterações de estado do protocolo sob teste e a determinação da cobertura de falhas de seus mecanismos de tolerância a falhas. No FIRMAMENT, são empregados recursos do sistema como o mecanismo de captura de mensagens klogd, que lê buffers de mensagens do kernel as repassa ao utilitário syslogd.

5 Experimentos de injeção de falhas

Os testes envolvendo a injeção de falhas no protocolo SCTP foram aplicados de maneira a exercitar os mecanismos de tolerância a falhas oferecidos pelo protocolo. Foram exercitadas as funcionalidades relacionadas com o controle de fluxo, verificação de associação, entrega seqüencial e controle de duplicação de mensagens.

Para cada experimento foi desenvolvido um *faultlet*, a ser usado na injeção de falhas. Foram escritos *faultlets* para obter os seguintes cenários de falha:

- descarte de pacotes.
- duplicação de pacotes;
- alteração dos *tags* de verificação.

Devido à limitação de espaço só mostraremos o *fautlet* escrito para o primeiro cenário.

5.1 Configuração de teste

Os experimentos foram realizados em máquinas idênticas com processador AMD Athlon XP 2000+, 512Mbytes de memória principal e controladores de rede modelo VIA Rhine II padrão IEEE 802.3u (Ethernet, 100 Mbit/s). Todas as máquinas utilizam o

kernel 2.6.20.3 do Linux, sendo que o FIRMAMENT foi carregado em apenas uma delas.

Para que fosse suportado o SCTP, foi necessário instalar a biblioteca lksctp (disponível em: < http://lksctp.sourceforge.net/>). Essa biblioteca permite o acesso dos usuários aos mecanismos do protocolo, que devem estar habilitados e compilados no kernel do Linux. O suporte ao SCTP foi incluído ao kernel como um módulo. Acompanha o código da biblioteca alguns programas de teste, que foram utilizados na geração e monitoramento do tráfego SCTP, sobre o qual as falhas foram injetadas.

5.2 Descarte de pacotes

Conforme apresentado anteriormente, o protocolo SCTP é capaz de detectar a perda de pacotes e reenviá-los de maneira a recuperar essa falha. Esse procedimento, entretanto, impõe uma queda de desempenho devido ao tempo que o cliente deve esperar até decidir por reenviar o pacote. Assim, para verificar o impacto desse mecanismo no desempenho da aplicação, escreveu-se um *faultlet* (figura 1) que descarta pacotes em percentuais pré-definidos de 10%, 20% e 40%. Esses percentuais foram aplicados a pacotes que continham um fluxo de dados, dois fluxos de dados e oito fluxos de dados.

```
; localizar o protocolo de transporte no cabeçalho IP
       SET    9     R0          ; deslocamento no pacote capturado
       READB  R0    R1
       SET    132   R0          ; seleciona os pacotes SCTP
       SUB    R0    R1
       JMPZ   R1    ACPSCTP     ; se o valor lido do cabeçalho for 132
                                ; desvia para aceitar ou descartar SCTP
       ACP                      ; aceita qualquer outro pacote
ACPSCTP:
       SET    100   R0          ; inicializa R0 para gerar número aleatório
       RND    R0    R1          ; gera valor aleat. entre -100 e 100 em R1
       SET    20    R0          ; define percentagem de descarte
       ADD    R0    R1          ;
       JMPN   R1    DESCARTE    ; se negativo então descarta pacote
       ACP                      ; se positivo então aceita pacote
DESCARTE:
       DRP
```

Figura 1 – Faultlet para descarte aleatório de pacotes SCTP

A figura 1 mostra o faultlet que permite testar se cada pacote capturado pertence ao protocolo SCTP, deixando passar pacotes de todos os demais protocolos. Quando encontra um pacote SCTP, o faultlet determina de acordo com uma taxa uniforme de distribuição de probabilidade se o pacote deve ser aceito ou descartado. Para cada percentual de descarte desejado deve ser escrito um novo faultlet.

Os vários *faultlets* e número de fluxos de dados foram combinados em nove configurações distintas. A estas, foram acrescentadas três configurações de referência, onde não foram injetadas falhas, resultando em um total de doze configurações.

Para avaliar o desempenho de cada configuração, foram efetuadas medições do tempo de processamento para enviar um total de 100 mensagens com 1500 bytes cada uma. Foram efetuadas 200 medições para cada configuração, para um índice de confiança de 95%. Na tabela 1 estão listados os tempos medidos nas configurações de referência (sem descartes), com um, dois e oito fluxos. A análise desses resultados mostra que o número de fluxos não afeta significativamente os tempos. A razão disso é

que, na maioria das transmissões, os vários fluxos estão sendo encapsulados no mesmo pacote. Para que obtivéssemos um ganho com o aumento do número de fluxos, seria necessário que houvesse vários rotas por onde os fluxos pudessem ser transportados em paralelo.

Tabela 1: tempo de processamento das configurações sem injeção de falhas

Número de fluxos	Tempo (seg)
1	2,805
2	2,786
8	2,923

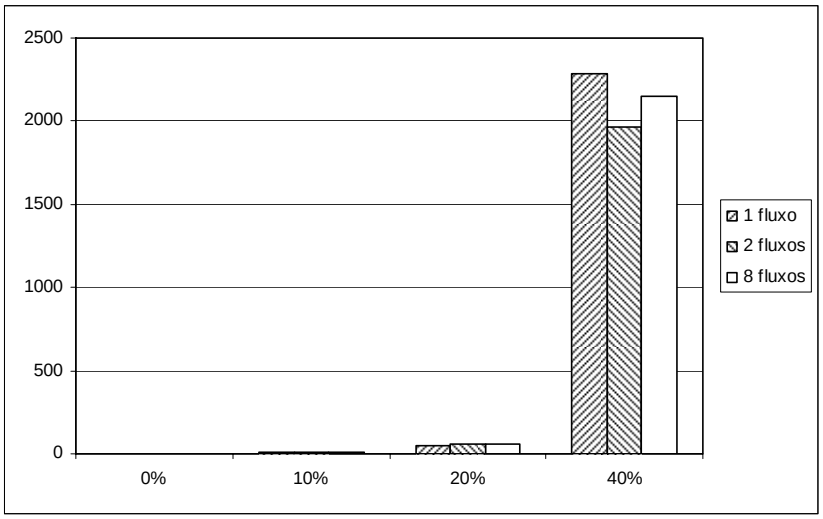
Na tabela 2 estão listadas as medidas efetuadas nas configurações onde foram aplicados os *faultlets* que descartam pacotes. Da mesma forma que no caso sem descarte, a análise dessas medidas mostra que os tempos permanecem, aproximadamente, os mesmos, independentemente do número de fluxos de dados usados para transportar os dados. O fato de poder transportar dados em fluxos independentes é apresentado como vantagem do protocolo SCTP. Essa característica deveria levar a um desempenho tanto maior quanto mais fluxos fossem usados. Entretanto, as medidas apresentadas na tabela 2 não confirmaram esse fato. Uma análise detalhada da forma como os *faultlets* foram projetados explica esse resultado: os *faultlets* foram projetados para descartar pacotes inteiros e não *chunks* dentro dos pacotes. Assim, sempre que um pacote é descartado, todos os fluxos sofrem a mesma perda. Note-se que a incapacidade em fornecer um melhor desempenho quando o número de fluxos é maior não é uma limitação do SCTP, mas sim da inexistência de rotas alternativas que pudessem ser usadas pelos diferentes pacotes.

Tabela 2: tempo de processamento das configurações com injeção de falhas

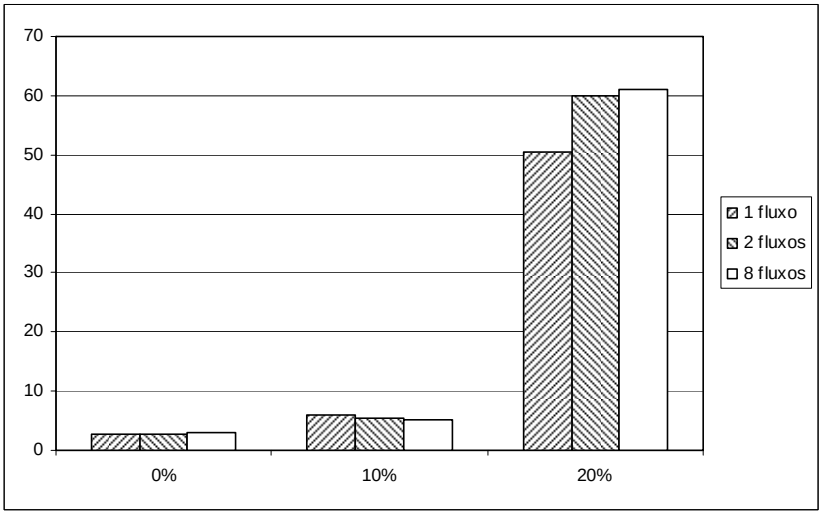
Número de fluxos	Tempo (seg) descarta 10%	Tempo (seg) descarta 20%	Tempo (seg) descarta 40%
1	6,095	50,501	2.287,204
2	5,313	60,011	1.961,677
8	5,212	61,083	2.152,372

O protocolo SCTP detecta a perda de pacotes através da temporização (*timeout*). O valor padrão para esse tempo é de 1 segundo. Esse tempo foi definido em função da aplicação para a qual o protocolo foi, originalmente, projetado (sinalização telefônica). Caso o protocolo SCTP seja usado para transportar informações de aplicações de dados, esse valor de temporização pode não ser adequado.

Os dados da tabela 1 e 2 estão representados na figura 2, onde pode-se verificar o aumento exponencial do tempo de processamento com o aumento do percentual de descarte de pacotes. No gráfico (a) da figura, pode-se verificar que o valor do tempo de processamento para um percentual de 40% de descarte é muito maior do que os outros, dificultando a visualização dessas outras medidas. No gráfico (b) da figura as medidas para 40% de descarte foram removidas, de maneira a evidenciar o comportamento das medidas restantes. Nesse gráfico, identifica-se, ainda, a tendência exponencial de crescimento com o aumento do percentual de descarte de mensagens.



(a)



(b)

Figura 2 – Tempo de processamento das configurações

5.3 Duplicação de pacotes

O protocolo prevê o envio de mensagens de reconhecimento da recepção de cada mensagem de dados. Assim, caso um pacote seja perdido, o transmissor identificará esse fato através do esgotamento do tempo de espera pela mensagem de reconhecimento e reenviará o pacote perdido. Entretanto, pode ocorrer da mensagem de reconhecimento chegar ao destino após ter esgotado o tempo de espera e, portanto, ter sido enviada uma repetição da mensagem. Com isso, o destinatário da mensagem de dados receberá dois pacotes idênticos, o que, segundo a especificação do protocolo SCTP, será detectado e corrigido (a instância do SCTP no receptor não entregará a duplicata para a aplicação).

Para verificar a robustez da implementação do SCTP, escreveu-se um *faultlet* capaz de duplicar todos os pacotes de dados enviados pelo cliente SCTP. Observou-se que a aplicação não percebia a duplicação de pacotes, uma vez que recebia um único exemplar de cada: a instância do SCTP no receptor filtrava as repetições do pacote. O

teste mostra que na ocorrência de duplicação de pacotes, a implementação do protocolo está em conformidade com a especificação.

5.4 Alteração dos *tags* de verificação

Durante o processo de estabelecimento de uma associação SCTP, os terminais envolvidos trocam informações que permitem a identificação da mesma. Essa identificação é feita através de *tags* de verificação, um para cada direção de fluxo (cliente para servidor e servidor para cliente), e identificam a associação de maneira inequívoca, de forma a impedir a recepção de dados de outras associações. Essa verificação ocorre na instância de implementação do SCTP que está sendo executada no receptor dos pacotes.

Para verificar a implementação do SCTP, escreveu-se um *faultlet* que injeta falhas no *tag* de verificação. Com isso, o pacote alterado não deverá ser reconhecido no receptor como sendo da associação corrente e, portanto, não será entregue para a aplicação. O procedimento para a aplicação do *faultlet* foi o seguinte: estabeleceu-se uma associação entre cliente e servidor; em seguida, o *faultlet* foi ativado; então, foram enviadas mensagens de dados.

Usando um dos programas monitores fornecidos com a biblioteca lksctp, observou-se que os pacotes SCTP chegavam ao destinatário. Entretanto, em conformidade com a especificação, as mensagens desses pacotes não eram entregues à aplicação, uma vez que o *tag* desses pacotes não correspondia ao *tag* registrado na associação estabelecida.

6 Conclusão e trabalhos futuros

Esse artigo mostrou que um injetor de falhas de comunicação operando no kernel do sistema operacional Linux é extremamente útil para o teste da implementação de protocolos de rede construídos sobre IP. Ganchos na pilha de protocolo permitem selecionar e atuar sobre todas as mensagens que fluem pela pilha. Uma ferramenta como FIRMAMENT, aplicada nos experimentos aqui reportados, pode tanto ser usada pelo desenvolvedor para o teste sob falhas de sua implementação como por um testador independente para teste de conformidade da implementação à especificação nas situações de falhas previstas. Injeção de falhas não substitui outros testes convencionais necessários, apenas facilita criar cenários de falhas usuais ao ambiente onde o protocolo vai operar. Tais cenários, uma combinação de carga de falhas e carga de trabalho, permitem o máximo de flexibilidade e controlabilidade para o teste sob falhas de comunicação.

A análise da implementação do protocolo SCTP através do uso de injetores de falhas não está encerrada. Os resultados obtidos possibilitaram verificar algumas das características de tratamento de erros da implementação do SCTP para Linux. Adicionalmente, no caso da perda de mensagens, foi possível avaliar o desempenho da implementação, o que levou ao questionamento do valor do tempo de espera para repetição da mensagem.

Em continuidade ao trabalho apresentado, alguns novos experimentos estão sendo propostos. O primeiro deles é a transmissão de dados através de rotas alternativas, que possibilitará avaliar a implementação no que diz respeito a essa

característica do protocolo. Além disso, será possível avaliar a eficiência com que o protocolo utiliza essa redundância.

Um segundo experimento será utilizar endereços alternativos para o estabelecimento dos fluxos de dados em uma associação SCTP. Dessa forma, serão possíveis, por exemplo, múltiplas conexões *http* independentes. O experimento consistirá em injetar falhas em alguns desses fluxos e observar o efeito nos outros fluxos. Segundo a especificação do protocolo, não deve haver interferência.

Um experimento semelhante ao dos endereços alternativos é o de perda de *chunks* de dados. Novamente, a perda de um *chunk* de dados de um fluxo não poderá interferir nos demais fluxos. A injeção de falhas permitirá verificar essa independência e, além disso, avaliar a queda de desempenho devido a essas falhas.

Como conseqüência das conclusões apresentadas nesse trabalho de que o desempenho piora significativamente quando ocorrem perdas de mensagens, um experimento natural é o de reduzir-se esse tempo e avaliar o impacto no desempenho. Como resultado desses experimentos espera-se poder sugerir uma temporização mais adequada ao uso do protocolo SCTP para o transporte de dados.

Finalmente, experimentos de injeção de falhas desta complexidade, conduzidos sobre protocolos reais implementados por terceiros permitirão um aprofundamento do conhecimento sobre as reais necessidades de testadores de implementações de protocolos. Com base no conhecimento prático adquirido durante os experimentos, ferramentas de injeção de falhas podem ser melhor definidas e projetadas para aumentar sua flexibilidade, funcionalidade, usabilidade e portabilidade.

7 Referências bibliográficas

Arlat, J.; Aguera, M.; Amat, L.; Crouzet, Y.; Laprie, J.; Fabre, J.; Martins, E.; Powell, D. (1990). Fault-injection for dependability validation: a methodology and some applications. IEEE Trans. on Software Eng., Special Issue on Experimental Computer Science, New York, vol. 16, n.2, p. 166-82, Feb.

Carreira, J.; Madeira, H.; Silva, J. G. (1998). Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers. IEEE Trans. on Software Eng., p. 125-135.

Chandra, R,; Lefever, R. M.; Joshi, K. R.; Cukier, M.; Sanders, W. H. (2004). A Global-State-Triggered Fault Injector for Distributed System Evaluation. IEEE Transactions On Parallel And Distributed Systems, v. 15, n. 7, July, p. 593-605

Dawson, S; Jahanian, F.; Mitton, T. (1996). ORCHESTRA: A Probing and Fault Injection Environment for Testing Protocol Implementations. Proceedings of IPDS'96. Urbana-Champaign, USA.

Drebes, R. J. (2005). FIRMAMENT: Um Modulo de Injeção de Falhas de Comunicação para Linux. Dissertação (Ciências da Computação) - Universidade Federal do Rio Grande do Sul

Drebes, R. J.; Silva, G. Jacques; Trindade, J. M. F.; Weber, Taisy Silva. (2006). A Kernel-based Communication Fault Injector for Dependability Testing of Distributed

Systems. Hardware and Software, Verification and Testing, First International Haifa Verification Conference, Revised Selected Papers. Lecture Notes in Computer Science. Berlin- Heidelberg: Springer-Verlag,. v. 3875, p. 177-190.

Han, S.; Shin, K. G.; Rosenberg, H. A. (1995). Doctor: An Integrated Software Fault-Injection Environment for Distributed Real-Time Systems. Second Annual IEEE Int'l Computer Performance and Dependability Symp., 1995, Erlangen. Los Alamitos. p. 204-13.

Hsueh, Mei-Chen; Tsai, T. K.; Iyer, R. K. (1997). Fault Injection Techniques and Tools. Computer, april, pp. 75-82

Jacques-Silva, G.; Drebes, R. J.; Trindade, J.; Weber, Taisy Silva; Pôrto, I. (2006). A Network-level Distributed Fault Injector for Experimental Validation of Dependable Distributed Systems. COMPSAC 2006. 30th Annual International Computer Software and Applications Conference. Chicago. IEEE Computer Society Press, 2006, v. 1. p. 421-428.

Looker, N.; XU, J. (2003). Assessing the dependability of OGSA middleware by fault injection. Proc. of the 22nd SRDS, p. 293–302, Florence, Italy.

Martins, E.; Rubira, C.M. F.; Leme, N. G. M. (2002). Jaca: A reflective fault injection tool based on patterns. Proceedings of the International Conference on Dependable Systems and Networks (DSN'02), pages 483–487.

Russel, R.; Welte, H. (2002). Linux net filter hacking HOWTO. 2002. Disponível em: <http://www.netfilter.org/documentation/>

Some, R. R. et al. (2001) A Software-Implemented Fault Injection Methodology for Design and Validation of System Fault Tolerance. IEEE Int'l Symposium on Dependable Systems and Networks, 2001, Göteborg (Sweden). p. 501-6.

Stewart, R. et al. (2000). Stream Control Transmission Protocol: RFC 2960. [S.1.]: Internet Engineering Task Force, Network Working Group.

Stewart, R. et al. (2004) Stream Control Transmission Protocol (SCTP) partial reability extension: RFC 4460. [S.1.]: Internet Engineering Task Force, Network Working Group.

Stewart, R. et al. (2006) Stream Control Transmission Protocol (SCTP) specification errata and issues: RFC 4460. [S.1.]: Internet Engineering Task Force, Network Working Group.

Stott, D. T.; Floering, B.,; Burke, D.,; Kalbarcyzk, Z.; Iyer, R. K. (2000). NFTAPE: a Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors. IEEE International Computer Performance and Dependability Symposium, pp. 91-100.

Modelo de um Ambiente para Descrição de Cenários Detalhados de Falhas

Ruthiano S. Munaretti, Taisy S. Weber

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brasil

{rsmunaretti,taisy}@inf.ufrgs.br

Abstract. *In message-based systems, fault injection approach causes faults in a controlled way. Thus, the behavior of these systems could be investigated on the presence of faults. However, each fault injector uses a different approach, causing several difficulties on usability of these tools. In this context, the present paper proposes an environment for detailed fault scenarios description, with the explanation of the main components of this environment, as well as the use of them in some fault injectors in the literature.*

Resumo. *Em aplicações baseadas em troca de mensagens, a técnica de injeção de falhas tem como objetivo provocar falhas de forma controlada. Assim, pode-se investigar o comportamento destes sistemas na presença de falhas. Entretanto, a existência de diversos injetores de falhas relacionados a este fim ocasionam uma certa dificuldade, inerente ao uso destes injetores, por adotarem abordagens distintas de funcionamento. Neste contexto, o presente trabalho apresenta um ambiente para descrição de cenários detalhados de falhas, abordando os principais elementos deste ambiente, bem como a aplicação dos mesmos em injetores de falhas existentes na literatura.*

1. Motivação

A realização de testes em sistemas computacionais é uma tarefa essencial e desafiadora. Para a execução desta tarefa, o uso de injetores de falhas é fundamental, visto que a ocorrência de falhas nestes sistemas é inevitável, o que pode ocasionar conseqüências desastrosas ao funcionamento dos mesmos. Ao mesmo tempo, a ocorrência de falhas em uma execução real do sistema tem uma probabilidade muito pequena de ocorrer, o que justifica ainda mais o uso de injetores de falhas, visando assim acelerar este processo em um determinado experimento.

Desta forma, injeção de falhas é uma técnica que tem como objetivo provocar falhas de forma controlada em um determinado sistema. A partir desta injeção, pode-se investigar o comportamento do sistema durante a presença de falhas, verificando o seu funcionamento, bem como mecanismos de tolerância a falhas implementados. Em comparação com a *modelagem analítica*, a injeção de falhas mostra-se mais adequada para a investigação de sistemas complexos [Arlat et al. 2003].

Considerando-se sistemas baseados em trocas de mensagens, as principais falhas são de *comunicação*. Por falhas de comunicação, entende-se todas as falhas que envolvem, no todo ou em parte, a rede de comunicação no qual o sistema em questão é executado. Assim, como exemplos de falhas de comunicação, podem ser destacados o colapso,

em um determinado momento, de algum *caminho* na rede de comunicação, assim como o atraso de mensagens e o colapso de nodos que formam a respectiva rede.

A proliferação de injetores de falhas para aplicações baseadas em troca de mensagens leva a uma dificuldade inerente ao uso dos mesmos. O principal problema está relacionado a *escolha* do injetor a ser utilizado para um dado experimento, visto que os mesmos possuem, em sua maior parte, abordagens distintas de funcionamento. Por este motivo, um esforço adicional é necessário nesta etapa, transcendendo assim a já complexa tarefa de elaboração de experimentos para testes de sistemas computacionais.

Neste contexto, o presente artigo aborda um modelo de ambiente para descrição de cenários detalhados de falhas, com foco em falhas de comunicação, conforme ilustrado na figura 1. O principal objetivo deste ambiente consiste em facilitar e expandir o uso dos diversos injetores de falhas existentes na literatura, aumentando assim a usabilidade dos mesmos. Algumas premissas para a realização deste artigo são abordadas nos itens a seguir.

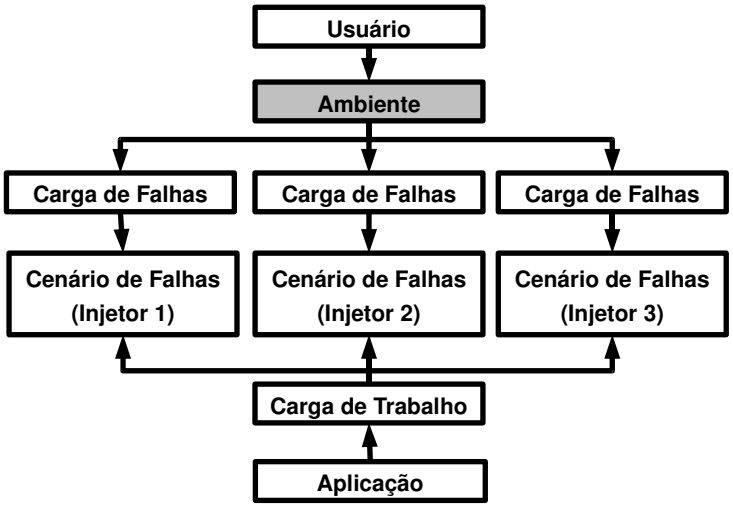


Figura 1. Ambiente para descrição de cargas de falhas.

- **Foco em falhas de comunicação:** na criação de cenários, ênfase será dada às falhas de comunicação, por serem as principais fontes de defeito no contexto de sistemas baseados em troca de mensagens.
- **Independência de injetor de falhas:** os procedimentos de criação de cenários de falhas serão os mesmos para quaisquer injetores de falhas que forem utilizados. Desta forma, cada cenário criado será convertido, de forma automática pelo ambiente, para a interface específica do injetor.
- **Elaboração de cenários *detalhados*:** o ambiente terá suporte a falhas múltiplas não simultâneas, onde uma seqüência de falhas é definida e executada pelo injetor específico, na ordem em que as mesmas forem definidas no respectivo ambiente.
- **Extensão/flexibilidade:** na construção do ambiente proposto, será prevista uma futura extensibilidade do modelo, de forma que o ambiente possa ser facilmente

adaptado para a construção de novos cenários de falhas, de acordo com a necessidade do projetista da aplicação.

A organização do artigo foi realizada da seguinte forma: a seção 2 apresenta uma visão geral sobre os injetores de falhas com foco em sistemas baseados em troca de mensagens, com ênfase aos injetores que mais se aproximam a este trabalho. A seção 3, por sua vez, descreve detalhadamente o modelo do ambiente, abrangendo seus elementos principais. Conclusões e detalhes sobre o andamento do trabalho são apresentados na seção 4.

2. Trabalhos Relacionados

Esta seção visa apresentar alguns dos principais injetores de falhas existentes na literatura. Primeiramente, serão descritos dois injetores desenvolvidos no Grupo de Tolerância a Falhas da UFRGS: FIONA e FIRMI, nas seções 2.1 e 2.2, respectivamente. Em seguida, serão abordados outros dois injetores conhecidos na literatura: DOCTOR (seção 2.3) e FAIL (seção 2.4).

2.1. FIONA

FIONA [Jacques-Silva et al. 2004] é um injetor de falhas com foco em sistemas distribuídos de *larga escala*. A abordagem utilizada no mesmo consiste na **instrumentação de código**, utilizando-se para isso da ferramenta JVMTI. O foco do injetor é o protocolo UDP, muito embora estejam previstas extensões para outros protocolos, tais como o TCP, por exemplo [Gerchman and Weber 2006].

A arquitetura *distribuída* de FIONA, assim como a local, também é constituída por três elementos: o **injetor principal** (responsável pelo gerenciamento do experimento), o **injetor de site** (que gerencia o conjunto de máquinas integrantes do respectivo site) e o **injetor local** (que aplicam as falhas propriamente ditas na aplicação alvo). A figura 2 ilustra esta arquitetura distribuída.

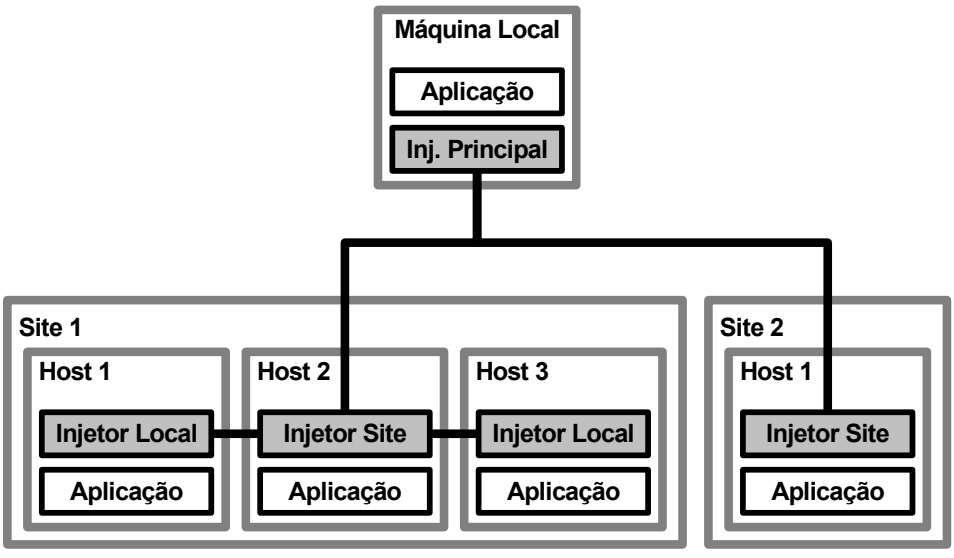


Figura 2. Arquitetura distribuída de FIONA [Jacques-Silva et al. 2004].

Para a criação de cenários de falhas, FIONA utiliza um arquivo de configuração, exigido somente no *injetor principal*. O funcionamento da carga de falhas na ferramenta FIONA ocorre a partir dos passos descritos a seguir.

- Ao interpretar o arquivo de configuração, um objeto `Fault` (que representa uma falha) é instanciado para cada falha especificada.
- Cada objeto `Fault`, criado no passo acima, será inserido no banco de falhas do experimento, denominado no injetor como `FaultBase`.
- Na execução do experimento, o banco de falhas (`FaultBase`) é consultado toda vez que ocorrer algum envio/recebimento de mensagem, a fim de verificar se a falha será ou não injetada na aplicação alvo.

2.2. FIRMI

FIRMI [Vacaro and Weber 2006] é uma ferramenta de injeção de falhas que emula cenários de falhas envolvendo JavaRMI (Remote Method Invocation). JavaRMI, um sistema baseado em objetos distribuídos, possui diversos pontos suscetíveis a falhas, principalmente se considerarmos a rede de comunicação no qual o mesmo é executado. Assim, a ferramenta FIRMI é utilizada para avaliar as aplicações alvo que são construídas sobre JavaRMI, além de avaliar também os mecanismos de tolerância a falhas existentes nestas aplicações.

A arquitetura de FIRMI é definida através de um *diagrama de classes*, ilustrado na figura 3. Como é possível visualizar nesta figura, os tipos de falhas são implementados através de *exceções*. Estas exceções são ativadas na aplicação alvo através de uma técnica de *instrumentação*. Esta instrumentação de código, por sua vez, é realizada a partir do *stub* e do *skeleton*, ambos componentes do sistema de comunicação JavaRMI, referentes ao cliente (que solicita as requisições) e ao servidor (que atende as requisições) da aplicação alvo, respectivamente.

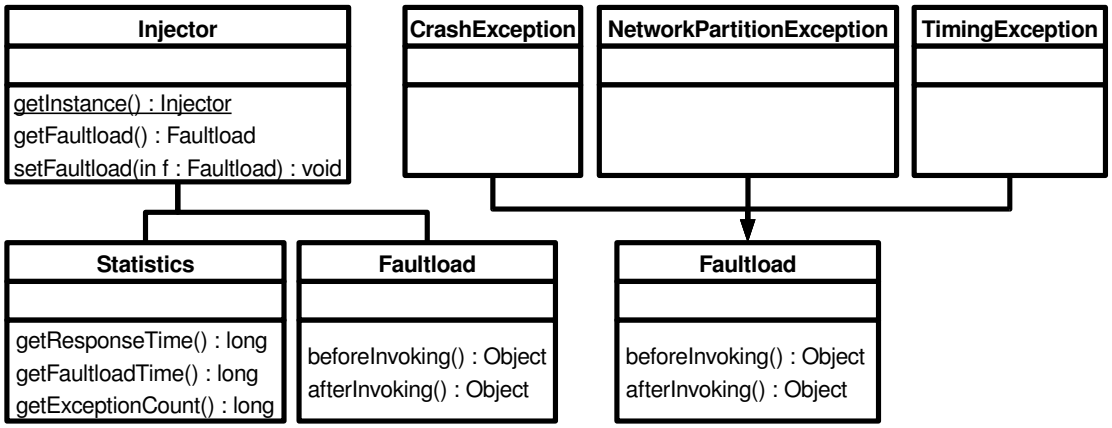


Figura 3. Diagrama de classes de FIRMI [Vacaro and Weber 2006].

Para a especificação de carga de falhas, FIRMI utiliza a classe `Faultload`. Assim, ao especificar uma carga, esta classe deve ser estendida, sendo a funcionalidade específica da mesma implementada através dos métodos `beforeInvoking()`

e `afterInvoking()`. Estes métodos são responsáveis pelas implementações das funcionalidades que ocorrem *antes* e *depois* da carga de falhas do injetor em si, respectivamente.

2.3. DOCTOR

A ferramenta DOCTOR [Han et al. 1995] consiste em um ambiente de injeção de falhas, com foco em sistemas distribuídos de tempo real. Seu objetivo inicial era injetar falhas na aplicação de tempo real HARTS [Shin 1991], aplicação em que DOCTOR foi utilizada de forma extensiva. Além do injetor em si, DOCTOR era formado por uma série de ferramentas auxiliares, tais como coletor de dados, gerador de cargas sintéticas de trabalho e interfaces gráficas (neste último, para interação com o usuário do injetor).

Referente à arquitetura, DOCTOR define uma série de componentes. Os principais componentes integrantes desta arquitetura são descritos nos parágrafos a seguir. A figura 4 ilustra a arquitetura esquematizada de DOCTOR.

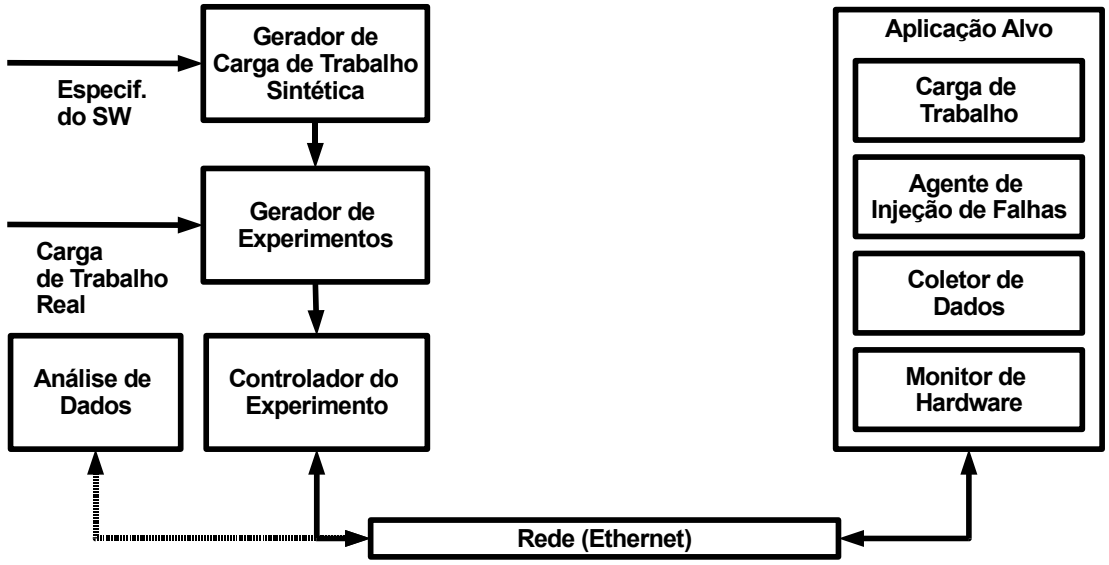


Figura 4. Arquitetura de DOCTOR [Han et al. 1995].

Primeiramente, o **Gerador de Experimentos** é o responsável pela obtenção da carga de trabalho da aplicação. Neste caso, podem ser usadas aplicações reais ou "artificiais" (obtidas a partir do **Gerador de Carga de Trabalho Sintética**). Outra função do Gerador de Experimentos é a de realizar a leitura do arquivo que descreve o experimento, que possui um formato próprio e contém informações sobre os tipos de falhas a serem consideradas, além do tempo em que as mesmas devem ser injetadas.

Com o experimento gerado, o **Controlador do Experimento** envia comandos para o **Agente de Injeção de Falhas** durante a execução deste experimento. Este agente, por sua vez, executa tais comandos injetando as falhas ou trocando o estado da carga de trabalho (neste caso, para os estados "wait", "start" ou "stop"). Ao mesmo tempo, o Agente de Injeção de Falhas realiza o *log* de suas atividades, enviando o mesmo aos componentes **Coletor de Dados** e **Monitor de Hardware**. Finalmente, o componente **Análise de Dados** é o responsável pela análise dos dados "pós-experimentos", com o intuito de gerar informações estatísticas a respeito dos testes executados.

2.4. FAIL/FCI

FAIL (**FA**ult **I**njection **L**anguage) [Hoarau and Tixeuil 2005] é uma linguagem para descrição de cenários de falhas, que trabalha sobre o injetor de falhas FCI (**F**AIL **C**luster **I**mplementation). Este injetor, por sua vez, foi desenvolvido para injeção de falhas em aplicações de *cluster* e *peer-to-peer* (P2P). O principal objetivo de FAIL/FCI é oferecer um mecanismo para a criação de cenários de falhas complexos, sejam eles probabilísticos ou determinísticos, sem a complexidade na criação destes cenários, inerentes aos injetores já existentes.

A arquitetura da ferramenta, ilustrada na figura 5 como um exemplo de um nodo em um sistema distribuído, é composta por três componentes: um **compilador**, uma **biblioteca** e um **daemon**. Ambos são descritos nos itens a seguir.

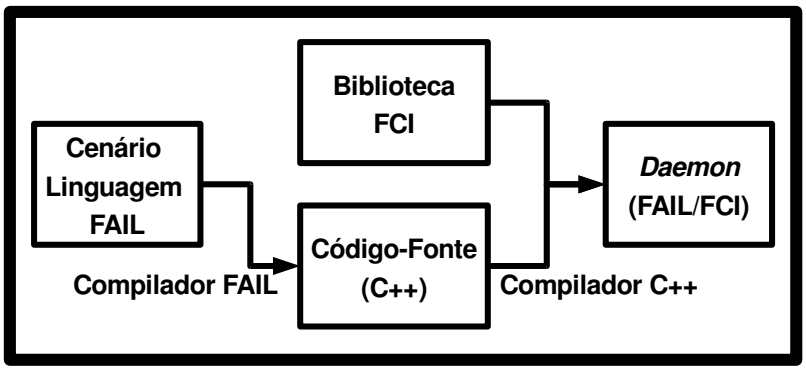


Figura 5. Arquitetura do injetor FCI [Tixeuil et al. 2006].

- **Compilador FCI:** responsável pela pré-compilação dos cenários de falhas escritos em FAIL, gerando códigos-fontes (no caso do FCI, em C++) que serão utilizados pela biblioteca FCI (descrita no próximo item), bem como arquivos de configuração para a realização do experimento.
- **Biblioteca FCI:** a partir dos arquivos gerados pelo compilador, a biblioteca FCI realiza a distribuição destes arquivos pelos nodos do sistema distribuído. Vale ressaltar que esta biblioteca distribui os arquivos como código-fonte *não compilado*, de forma a manter a *heterogeneidade* entre os nodos que formam o *cluster* do experimento.
- **Daemon FCI:** componente presente em cada nodo do *cluster*, o daemon é encarregado de realizar as compilações dos fontes em cada nodo, bem como realizar a instrumentação de código, que trata-se da injeção de falhas em si na aplicação alvo.

De todos os injetores analisados neste trabalho, o FAIL/FCI é o que se apresenta de forma mais adequada nos quesitos de criação de cenários de falhas e de extensibilidade, uma vez que a linguagem FAIL é expressiva e poderosa, permitindo a criação de cenários de falhas complexos com simplicidade, bem como uma extensão facilitada de um determinado modelo de falhas implementado. Entretanto, a linguagem FAIL possui a limitação de funcionar apenas com o injetor FCI, que por sua vez é limitado aos ambientes de *clusters* e redes *peer-to-peer*, conforme já citado anteriormente. Além disso, outra

limitação é relacionada ao modelo de falhas implementado pela ferramenta, envolvendo apenas falhas relacionadas a colapso.

3. Modelo do Ambiente

Um ambiente de cenários de falhas deve refletir todos os casos possíveis de falhas, de acordo com o tipo de aplicação alvo desejado. Neste sentido, o modelo descrito neste artigo foi elaborado visando a divisão em elementos, modularizando as funcionalidades do ambiente, bem como promovendo a extensibilidade do mesmo. A figura 6 ilustra uma visão macro da arquitetura do ambiente, sendo cada elemento descrito nos itens a seguir.

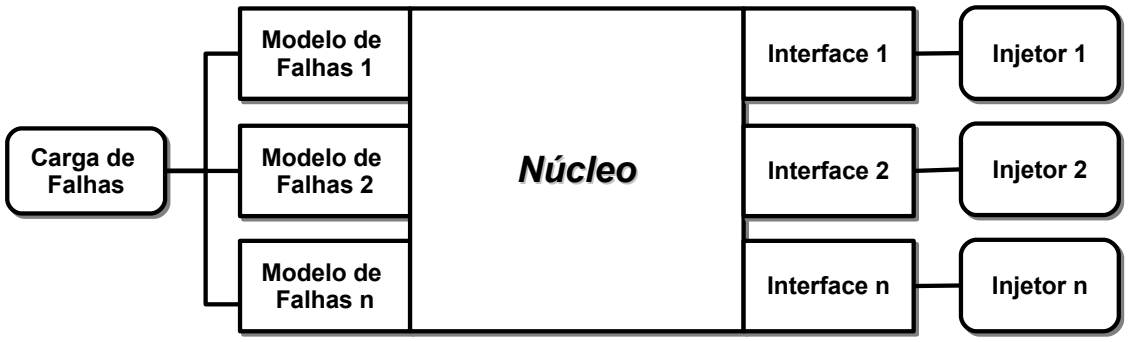


Figura 6. Arquitetura do Ambiente

- **Carga de Falhas:** representando uma *entrada de dados* ao ambiente pelo usuário, a carga de falhas visa especificar *quais* falhas estarão ativas em um dado experimento, bem como qual a *configuração* de cada falha no respectivo experimento. É análoga às cargas de falhas realizadas em injetores específicos, com a vantagem de ser genérica, ou seja, não ser restrita às peculiaridades de uma determinada ferramenta.
- **Modelo de Falhas:** define um conjunto de falhas admitidas por uma determinada aplicação. Assim, são especificados os tipos de falhas que estarão disponíveis ao usuário em seu experimento. Os tipos definidos neste modelo delimitarão a funcionalidade da carga de falhas. Assim como a carga de falhas, esta também é uma entrada de dados definida pelo usuário, sendo que o ambiente fornece uma forma *intuitiva* para que o usuário especifique modelos de falhas com facilidade.
- **Núcleo:** sendo o elemento de mais baixo nível e, conseqüentemente, o mais importante do ambiente, o *núcleo* define uma série de *falhas primitivas* existentes em sistemas baseados em troca de mensagens. Logo, o principal objetivo consiste em mapear a carga de falhas, definida pelo usuário, às falhas primitivas existentes, iniciando-se assim a transformação da carga de falhas genérica para a carga de falhas específica de cada injetor utilizado.
- **Interface:** elemento externo ao núcleo, a interface tem por objetivo *finalizar* a transformação da carga de falhas genérica para a carga de falhas específica do injetor utilizado. Para isso, o ambiente define a interface como uma outra entrada de dados definida pelo usuário e, assim como o modelo de falhas, é fornecida uma forma *intuitiva* para que o usuário especifique a *forma* pelo qual o seu injetor específico trabalha.

- **Injetor:** finalmente, o elemento *injetor* representa um injetor de falhas específico que o usuário deseja utilizar no ambiente. O ambiente permite a utilização de vários injetores simultaneamente, sendo que os mesmos recebem como entrada suas respectivas cargas de falhas específicas, geradas pelas interfaces descritas no item anterior.

Desta forma, pode-se observar que o ambiente possui três entradas de dados, correspondentes a *Carga de Falhas*, ao *Modelo de Falhas* e à *Interface*. Referente a estes dois últimos, percebe-se que os mesmos são integrados ao *núcleo* do ambiente, sendo descritos pelo usuário utilizando a abordagem de *plugins*. Assim, o ambiente torna-se modularizado e extensível, facilitando ainda mais o seu uso.

Nas subseções seguintes, estes três elementos, juntamente com o núcleo, serão descritos em detalhes, abrangendo o funcionamento interno de cada um deles. Para esta descrição, será utilizada uma abordagem *bottom-up*. Assim, será descrito primeiramente o *núcleo*, por ser o elemento de mais baixo nível. Logo após, será abordado o *Modelo de Falhas*, a *Interface para Injetores Específicos* e, finalmente, a descrição da *Carga de Falhas*.

3.1. Núcleo

O *núcleo*, por ser o elemento de mais baixo nível do ambiente, define todas as operações básicas suportadas pelo mesmo. Logo, o núcleo será o responsável pela delimitação das falhas a serem implementadas, criando-se assim o *escopo* das falhas que poderão ser injetadas em um determinado experimento. Visando principalmente questões de desempenho, o núcleo implementado por este ambiente adotará uma abordagem simplificada, sendo assim formado por um conjunto pequeno de primitivas.

O foco do ambiente, conforme descrito na seção 1, está voltado a falhas de comunicação. Sistemas baseados em troca de mensagens, por terem a sua execução realizada em uma rede de computadores, são compostos basicamente por três componentes fundamentais, a saber: **nodos**, **caminhos** e **pacotes**. Estes componentes, conceituados nos itens que seguem, serão utilizados no núcleo do ambiente para a realização efetiva da injeção de falhas. Uma ilustração dos mesmos, de forma simplificada, é mostrada na figura 7.

- **Nodos:** formam as *unidades de execução* do sistema em questão, possuindo estado interno próprio. Assim, dependendo do contexto, um nodo pode representar um **emissor**, um **receptor** ou **ambos**.
- **Caminhos:** representam as *unidades de comunicação* entre dois nodos quaisquer do sistema. Desta forma, a troca de dados entre os nodos do sistema só pode ser realizada através de caminhos. Além disso, caminhos possuem uma velocidade de propagação específica, que indica o tempo no qual um determinado dado leva para ser transportado de um nodo do sistema para outro.
- **Pacotes:** os pacotes correspondem às *unidades de dados* existentes no sistema. Portanto, quando um determinado nodo do sistema deseja transmitir ou receber conteúdos de algum outro nodo, estes conteúdos são armazenados em um ou mais pacotes, para assim serem transportados a partir do caminho existente. Logo, um determinado pacote, durante a execução do sistema, pode estar no contexto do

nodo, do caminho ou em ambos (neste caso, representando um pacote *em envio* ou *em recebimento*), como ilustrado na figura 7.

- **Rede:** finalmente, o componente rede engloba uma união entre todos os componentes acima, sendo os nodos interligados através dos caminhos, com os respectivos pacotes fluindo entre eles. Desta forma, o componente em questão representa o sistema como um todo.

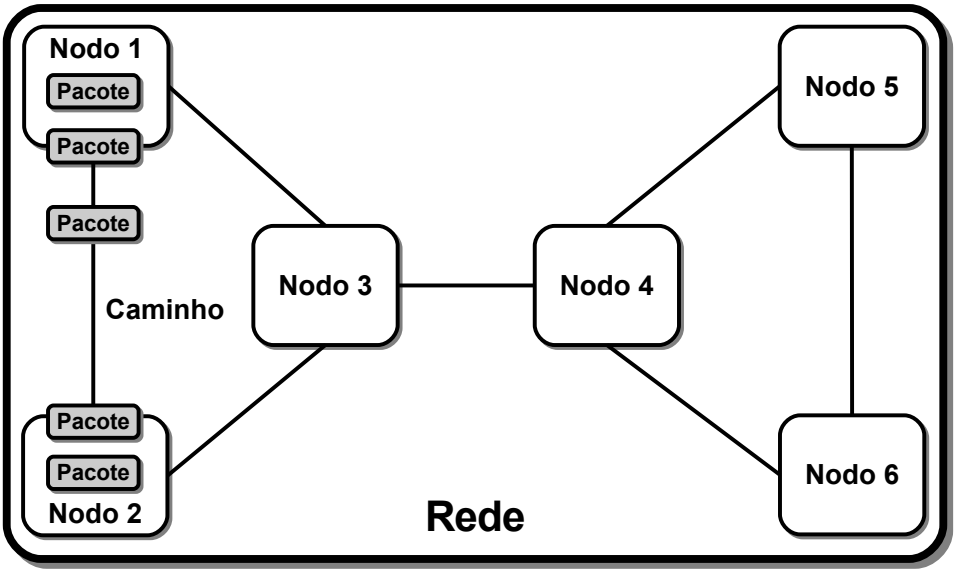


Figura 7. Componentes fundamentais de sistemas baseados em troca de mensagens

Além disso, outra abordagem adotada pelo núcleo do ambiente referem-se às *falhas primitivas* suportadas pelo ambiente. Por falha primitiva, entende-se um tipo de falha comumente encontrada em sistemas baseados em troca de mensagens, podendo levar o respectivo sistema a um estado *inconsistente*. Desta forma, o componente *Modelo de Falhas* (que será descrito na subseção 3.2) poderá fazer uso destas falhas primitivas, deixando-se assim o ambiente *independente* de um modelo de falhas específico.

No contexto do ambiente aqui descrito, as falhas primitivas serão aplicadas sobre os **pacotes** do sistema em questão, afetando, conseqüentemente, os **nodos** e **caminhos**. Isso ocorre porque o ambiente é voltado a falhas de **comunicação**. Vale lembrar que estes pacotes podem estar em contexto de nodo, caminho ou ambos, o que determinará sobre qual dos dois componentes (neste caso, nodo ou caminho) a respectiva falha primitiva será aplicada. Assim, foram adotadas as falhas primitivas apresentadas nos itens a seguir.

- **Reinício:** o componente em questão é *reiniciado*, tendo assim sua configuração retornada ao seu respectivo estado inicial. Nos contextos de nodo e caminho, isto significa que todos os pacotes serão descartados, sem possibilidade de recuperação dos mesmos.
- **Parada:** o componente tem a sua execução *suspensa*, podendo ser *retomada* em algum momento posterior do experimento. Em nodos, esta falha primitiva corresponde a parada no envio e recebimento de pacotes, enquanto que em caminhos, indica a parada na propagação de pacotes. Ao retomar a execução, os pacotes que

estavam presentes nos respectivos componentes **não** são perdidos, seguindo-se assim os destinos já definidos anteriormente para os mesmos.

- **Perda de pacotes:** como o nome indica, o componente sofrerá o descarte de um conjunto dos pacotes existentes no contexto do mesmo. Este descarte pode ocorrer de forma *determinística* ou *probabilística*. O descarte determinístico é baseado em um evento fixo/previsível do sistema, enquanto que o probabilístico é baseado em um percentual de pacotes que devem ser descartados durante o experimento, independente do conteúdo dos mesmos.

- **Atraso:** nesta falha primitiva, todos os pacotes presentes em um dado componente sofrem um atraso, parametrizável pelo usuário do ambiente. Assim como na perda de pacotes, este atraso pode ser *determinístico* (a partir de um valor fixo de atraso) ou *probabilístico* (a partir de uma distribuição aleatória).

Referente à arquitetura do ambiente, a mesma é definida através de uma *hierarquia de classes*. Esta técnica é adequada para a construção do núcleo, uma vez que a mesma permite delimitar, a partir de um componente genérico, todas as funcionalidades necessárias - neste caso, as falhas primitivas. Desta forma, fica a cargo de componentes específico a implementação destas funcionalidades, que não podem fugir ao escopo definido no componente genérico.

No caso do ambiente, é definida uma interface **Component**, que define os métodos correspondentes às falhas primitivas definidas nos itens anteriores, a saber: **restart()**, **stop()**, **dropPackets()** e **delayPackets()**, respectivamente. No contexto do núcleo, a classe **Network** engloba um conjunto de objetos da classe **Component**. Além disso, temos as classes específicas **Node** e **Path**, que implementam os métodos de **Component**, além da classe **Packet**, abrangendo assim todos os componentes fundamentais abordados. O diagrama UML desta estrutura de classes pode ser visualizado na figura 8.

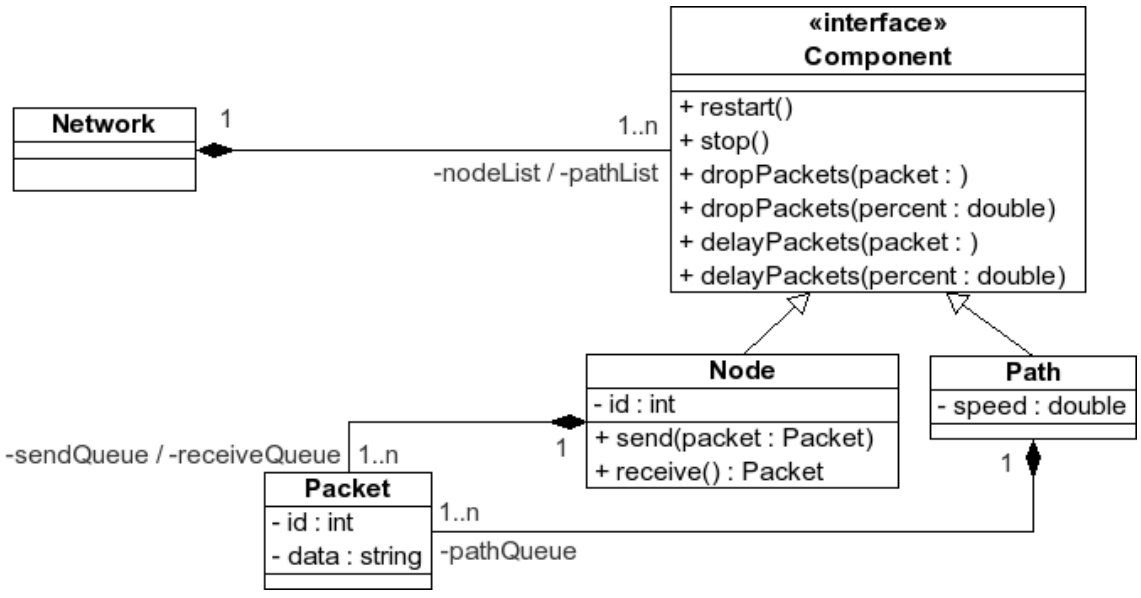


Figura 8. Diagrama UML do núcleo do ambiente

3.2. Modelo de Falhas

Conceitualmente, um modelo de falhas define um conjunto de falhas admitidas por uma determinada aplicação. Assim, cabe ao experimento emular falhas dentro deste modelo, contendo assim todos os tipos de falhas necessários para a sua execução. Desta forma, é possível saber, no dado experimento, quais tipos de falhas poderão ser injetadas e quais tipos não serão tratados.

Considerando sistemas baseados em troca de mensagens, existem diversos modelos de falhas disponíveis na literatura, dentre os quais podem ser destacados os modelos definidos por Cristian [Cristian et al. 1986] e Birman [Birman 1996]. Cristian define as falhas de colapso, omissão, temporização e bizantinas. Birman, por sua vez, define as falhas de colapso, parada segura (*fail-stop*), omissão de envio, omissão de recepção, rede, particionamento de rede, temporização e bizantinas.

No contexto do ambiente, é utilizada uma abordagem simplificada para a especificação de um modelo de falhas, baseada em *script*. Esta abordagem é ilustrada na figura 9 e explicada nos parágrafos subseqüentes.

```
<Component>.<Fault Label> { <Primitive Fault> [,;] <PrimitiveFault> [,;] ... }
<Component>.<Fault Label> { <Primitive Fault> }
...
```

Figura 9. Abordagem *script* para especificação de modelos de falhas

A linguagem *script* utilizada, conforme figura 9, é formada por três construções. A construção <Component> representa o **componente** para o qual se deseja especificar o tipo de falha, podendo ser Node ou Path. <Fault Label> indica um **rótulo** para o tipo de falha, sendo diretamente especificado pelo usuário do script. Finalmente, <Primitive Fault> especifica uma falha primitiva a ser adicionada no tipo de falha a ser especificado, podendo ser um dos métodos pertencentes a classe **Component**, vista na subseção 3.1.

```
Node.Crash { restart() , stop() }
Node.Omission { dropPackets(packet) ; dropPackets(percent) }
Node.Timing { delayPackets(packet) ; delayPackets(percent) }
Node.Byzantine {
    stop() ; dropPackets(packet) ; delayPackets( Uniform(percent) )
}
```

Figura 10. Exemplo para especificação de modelos de falhas

Ademais, esta linguagem *script* é utilizada a partir de um arquivo de configuração, sendo uma linha para cada tipo de falha. O tipo de falha é acoplado ao respectivo componente, através do caractere ".". As falhas primitivas referentes a cada tipo de falha são delimitados pelos caracteres "{" e "}". Neste escopo de falhas primitivas, cada elemento (que representa uma falha primitiva) é separado pelos caracteres "," ou ";", representando os operadores lógicos "E" (execução de *todas* as falhas primitivas relacionadas no escopo)

e “OU” (execução de apenas *um* das falhas primitivas relacionadas no escopo, escolhido de forma aleatória), respectivamente.

Desta forma, mesmo sendo simplificada, a linguagem criada permite a especificação de modelos elaborados de falhas, refletindo de maneira adequada os conceitos indicados nos mesmos. Para efeito de exemplo, a figura 10 ilustra um exemplo de modelo de falhas especificado a partir desta linguagem - neste caso, é ilustrada a especificação do modelo proposto por Cristian.

3.3. Interface para Injetores Específicos

Este elemento visa mapear um cenário de falhas criado no ambiente no formato de um determinado injetor. Neste formato, o cenário é chamado *cenário específico*, correspondendo assim à carga efetiva de falhas ao injetor específico, servindo como dados de entrada ao mesmo. Desta forma, objetiva-se oferecer a compatibilidade necessária aos injetores de falhas existentes.

```
<Fault Label 1> : <Command>
<Fault Label 2> : <Command>
...
<Fault Label n> : <Command>
```

Figura 11. Script de mapeamento - modelo de falhas/especificação do injetor

Assim como o modelo de falhas, descrito na subseção 3.2, este elemento é descrito pelo usuário seguindo-se a abordagem de *plugin*. Em outras palavras, o usuário do ambiente especifica a *forma* pelo qual a especificação do injetor específico é **mapeada** ao modelo de falhas já definido no ambiente. Para isso, é utilizado um *script de mapeamento*, cuja sintaxe pode ser visualizada na figura 11.

Neste script, cada linha representa um mapeamento. Este mapeamento, separado pelo caractere “:”, corresponde a um modelo de falhas do ambiente (representado por “<Fault Label>” pelo respectivo comando a ser chamado no injetor específico (representado por “<Command>”. Desta forma, é possível realizar a ligação entre o ambiente e o injetor de forma simples, facilitando assim a execução dos experimentos.

```
Node.Crash : UdpCrashFault
Node.Omission : UdpOmissionFault
```

(a) FIONA

```
Node.Crash : halt
Node.Omission : stop, continue
```

(b) FAIL/FCI

Figura 12. Exemplos de scripts de mapeamento

Vale ressaltar que o ambiente permite a inclusão de vários injetores de falhas simultaneamente a uma execução de um dado experimento. Por este motivo, são ilustrados dois exemplos de scripts de mapeamento na figura 12. A figura 12(a) representa um mapeamento para o injetor FIONA [Jacques-Silva et al. 2004] (um injetor de falhas para aplicações de rede), enquanto que a figura 12(b) ilustra um mapeamento para o injetor

FAIL/FCI [Hoarau and Tixeuil 2005] (um injetor para aplicações do tipo *grid*). Em ambos os casos, o modelo de falhas utilizado refere-se ao proposto por **Cristian** e, logo, são utilizados aqui os rótulos definidos na figura 10.

3.4. Carga de Falhas

A partir da carga de falhas, o usuário do ambiente poderá especificar exatamente quais as falhas estarão presentes em seu experimento, bem como as configurações que cada uma delas possuirá. Entre os elementos descritos, este é considerado o de mais alto nível, uma vez que o mesmo faz uso direto do modelo de falhas definido no ambiente.

Seguindo-se a abordagem adotada nos demais elementos do ambiente, a carga de falhas é realizada a partir de um *script* simplificado. A sintaxe deste script busca a *generalidade*, de forma a permitir uma especificação adequada de cenários de falhas, no contexto de falhas de comunicação. A figura 13 ilustra a estrutura utilizada neste script.

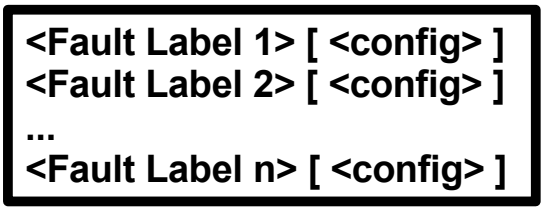


Figura 13. Estrutura utilizada no script para carga de falhas

De acordo com esta estrutura, pode-se visualizar que a mesma permite a definição de uma *seqüência* de falhas (sendo uma ocorrência por linha), caso seja necessário. Vale ressaltar que o usuário do ambiente não precisa, necessariamente, utilizar todos os tipos (estrutura "<FaultLabel>") definidos no modelo de falhas em um experimento, bem como os comandos de configuração (estrutura "<config>", que depende do tipo de falha utilizada). Entretanto, pelo menos um tipo deve ser utilizado, de forma que a injeção de falhas, no experimento, possa ocorrer de forma efetiva.

Para exemplificar a utilização deste script, é exibida na figura 14 uma carga de falhas genérica. Nesta carga, ocorre uma seqüência de duas falhas. Na primeira, ocorre um colapso de nodo, sem a necessidade de configurações adicionais. Já na segunda falha, ocorre uma falha de omissão em um caminho, com uma configuração adicional representada como "0.1", indicando uma probabilidade uniforme de perda em 10% dos pacotes. Em ambos os casos, é assumido o modelo de **Cristian**, conforme figura 10.

Figura 14. Exemplo de carga de falhas genérica

4. Conclusões

Este artigo apresentou um modelo de ambiente para descrição de cenários detalhados de falhas, com ênfase em falhas de comunicação. Foram abordados os elementos principais

deste ambiente, com uma descrição detalhada de cada um deles. Além disso, nos elementos de mais alto nível (ou seja, onde ocorre interação direta com o usuário do ambiente), foram ilustrados exemplos de caso de utilização, envolvendo alguns injetores de falhas existentes na literatura.

A partir do modelo do ambiente, foi constatado que o mesmo apresenta uma arquitetura adequada para a realidade de cenários de falhas, com pontos de extensibilidade bem definidos. Vale ressaltar também a facilidade de uso do ambiente, a partir de linguagens *script* simplificadas. Com isso, o uso do ambiente torna-se adequado para experimentos que envolvam sistemas baseados em troca de mensagens, independente da complexidade dos mesmos.

Referências

Arlat, J., Crouzet, Y., Karlsson, J., Folkesson, P., Fuchs, E., and Leber, G. H. (2003). Comparison of Physical and Software-Implemented Fault Injection Techniques. *IEEE Transactions on Computers*, 52:1115–1133.

Birman, K. (1996). *Building Secure and Reliable Network Applications*. Manning Publications, Co, Greenwich.

Cristian, F., Aghili, H., and Strong, R. (1986). *Clock Synchronization in the Presence of Omissions and Performance Faults, and Processor Joins*. In *16th International Symposium on Fault Tolerant Computing Systems*.

Gerchman, J. and Weber, T. S. (2006). *Emulando o Comportamento de TCP/IP em um Ambiente com Falhas para Teste de Aplicações de Rede*. In SBC, editor, *VII Workshop de Testes e Tolerância a Falhas - WTF2006*, pages 41–52, Curitiba, PR.

Han, S., Shin, K., and Rosenberg, H. (1995). *DOCTOR: An Integrated Software Fault Injection Environment for Distributed Real-Time Systems*. In *Int. Computer Performance and Dependability Symposium. (IPDS'95)*, pages 204–213, Erlangen, Germany. IEEE Computer Society Press.

Hoarau, W. and Tixeuil, S. (2005). *A Language-Driven Tool for Fault Injection in Distributed Systems*. In *Proc. of the 6th IEEE/ACM Intl. Workshop on Grid Computing*, pages 194–201, Grand Large, França.

Jacques-Silva, G., Drebes, R., Gerchman, J., and Weber, T. (2004). *FIONA: A Fault Injector for Dependability Evaluation of Java-Based Networks*. In *Proc. of the 3rd IEEE Intl. Symposium on Network Computing and Applications*, pages 303–308, Cambridge, MA.

Shin, K. (1991). *HARTS: A Distributed Real-Time Architecture*. *IEEE Computer*, 24(5):25–35.

Tixeuil, S., Hoarau, W., and Silva, L. (2006). *An Overview of Existing Tools for Fault-Injection and Dependability Benchmarking in Grids*. Technical Report TR-0041, CoreGRID (http://www.coregrid.net).

Vacaro, J. C. and Weber, T. S. (2006). *FIRMI: Um Injetor de Falhas para a Avaliação de Aplicações Distribuídas baseadas em RMI*. In SBC, editor, *VII Workshop de Testes e Tolerância a Falhas - WTF2006*, pages 159–170, Curitiba, PR.

XTool: Uma Ferramenta de Teste Baseado em Defeitos para Esquemas de Dados

Igor F. Nazar[1], Maria Claudia F. P. Emer[2], Silvia R. Vergilio[1], Mario Jino[2]

[1]DInf –UFPR, CP: 19081, CEP: 81.531-970 – Curitiba – PR – Brasil

[2]DCA – FEEC – UNICAMP, CP:6101, CEP: 13.083-970 – Campinas – SP - Brasil

{igor,silvia}@inf.ufpr.br, {mcemer,jino}@dca.fee.unicamp.Br

Abstract *This work describes a tool, named XTool, which tests data schemas considering previously defined classes of faults. XTool automatically generates test data composed by data instances and queries to these instances. The implemented approach allows the test of different kind of schemas. XTool contributes to improve the quality of the software applications, by ensuring the integrity of the data that it manipulates and that is defined by the schema under test.*

Resumo. *Este trabalho descreve a ferramenta, denominada XTool, que testa esquemas de dados considerando classes de defeitos previamente definidas.* A *XTool gera automaticamente dados de teste formados por instâncias de dados e consultas a essas instâncias. A abordagem implementada pela ferramenta pode ser utilizada em diferentes tipos de esquemas. O objetivo do teste realizado pela XTool é contribuir para melhorar a qualidade das aplicações de software, assegurando a integridade dos dados que ela manipula e que são definidos pelo esquema em teste.*

1. Introdução

O uso de sistemas baseados em computação exige o desenvolvimento de software com alto padrão de qualidade. Um aspecto importante para garantir a qualidade de um sistema está relacionado à integridade das informações que a aplicação manipula, tanto para a integração entre aplicações quanto para armazenamento de dados. Esses dados, em geral, são definidos por um esquema, que contém regras referentes à estrutura lógica dos dados permitida. No contexto de integração de sistemas, destaca-se a linguagem XML (*eXtensible Markup Language* (W3C 2006a)) utilizada para padronização de documentos. Um exemplo de esquema XML bastante utilizado é o XML *Schema* (W3C 2006b). No contexto de armazenamento de dados, destacam-se bases de dados relacionais associadas a esquemas descritos por meio do Modelo Entidade-Relacionamento (MER) (Chen 1976).

Geralmente os dados são validados por *parsers* de acordo com o esquema. Entretanto, para garantir a qualidade dos dados isso não é suficiente, pois um esquema pode estar incorreto com relação à especificação e escrito de maneira sintaticamente correta, ou seja, ser válido. Um esquema incorreto pode validar dados incorretos que quando recebidos pela aplicação produzem uma falha, ou seja, o esquema incorreto permite que dados incorretos sejam considerados válidos ou o contrário.

Uma forma de evitar isto é realizar o teste do esquema, após a sua especificação ou atualização. Na literatura, a maioria dos trabalhos existentes se dedica somente a testar as aplicações. No contexto de esquemas de dados relacionais, eles visam à

geração de dados, ao teste do projeto e da aplicação de base de dados (Chan e Cheung 1999, Chays et al. 2000, Chays e Deng 2003, Deng et al. 2005, Kapfhammer e Soffa 2003, Suárez-Cabal e Tuya 2004, Zhang et al. 2001) e quando consideram o esquema é somente como fonte de informações para a obtenção de dados de teste. (Chan et al. 2005, Robbert e Maryanski 1991). No contexto de esquemas XML, a maioria dos trabalhos tem como objetivo testar a interação de componentes que se comunicam utilizando a linguagem XML (Lee e Offutt 2001) e serviços Web (Offutt e Xu 2004, Xu et al. 2005).

Todos esses trabalhos não têm como foco o teste de esquema. São poucos os trabalhos que se dedicam a esse tema. Outro aspecto a ser ressaltado é que é mais raro ainda encontrar ferramentas que auxiliem na realização dessa tarefa. Nesse contexto, destacam-se as ferramentas *RDBTool* (*Relational Data-Base Testing Tool*) (Aranha et al. 2000) e *XTM* (*Tool for XML Schema Testing Based on Mutation*) (Franzotte e Vergilio 2006). A primeira ferramenta permite o teste estrutural em esquemas de base de dados relacional, que visa testar associações entre definições e usos de atributos em um esquema. A *XTM* apóia o teste de mutação em esquemas escritos em XML *Schema*. Essa ferramenta implementa operadores de mutação que provocam ligeiras modificações no esquema em teste, gerando esquemas mutantes. Os dados de teste são gerados para provocar um comportamento diferente entre os esquemas mutantes e o esquema original. Esse comportamento diferente está em validar ou não os dados.

Ambas as ferramentas são úteis e facilitam a aplicação de uma abordagem de teste para os respectivos contextos focalizados. Entretanto, elas apresentam algumas limitações tais como: 1) a abordagem implementada por estas ferramentas não pode ser diretamente aplicada a outros tipos de esquemas, tais como objeto-relacional, esquemas DTD (*Document Type Definition*), etc. 2) elas não permitem a geração automática de dados de teste, uma das mais custosas tarefas e consumidora de esforço para aplicação de uma abordagem ou critério de teste.

Para lidar com essas limitações, este artigo descreve a ferramenta *XTool* (*XML and Relational Database Schema Testing Tool*), que implementa uma abordagem baseada em defeitos genérica denominada Análise de Instâncias de Dados Alternativas (AIDA). Essa abordagem é baseada em defeitos porque considera classes de defeitos comumente presentes em esquemas. É genérica porque está baseada em um modelo formal que permite a representação de diferentes tipos de esquemas (Emer et al. 2007a). Na abordagem de teste AIDA, uma instância de dados associada ao esquema em teste sofre alterações simples gerando instâncias de dados alternativas. Essas instâncias de dados são geradas com base em padrões especificados nas classes de defeitos previamente identificadas. Consultas às instâncias de dados alternativas também são geradas a partir de padrões definidos nas classes de defeitos. As instâncias de dados representam os possíveis defeitos no esquema e as consultas são capazes de revelar esses defeitos. Portanto, a abordagem é denominada Análise de Instâncias de Dados Alternativas devido ao uso de instâncias de dados alternativas para detectar defeitos em esquemas de dados.

O artigo é organizado como segue: a Seção 2 descreve alguns trabalhos relacionados; a Seção 3 descreve a ferramenta *XTool*, a abordagem implementada e suas principais funcionalidades; a Seção 4 apresenta um exemplo de utilização da *XTool* no teste de um esquema XML; a Seção 5 contém resultados de estudos de caso realizados com o apoio da *XTool*; a Seção 6 apresenta as conclusões e trabalhos futuros.

2. Trabalhos Relacionados

O teste em aplicações de base de dados envolve aspectos relacionados à correção, tais como: comportamento da aplicação em relação à especificação; esquema da base de dados em relação aos dados do mundo real modelados; acurácia na base de dados, segurança e privacidade; e execução correta de operações de inserção, atualização e exclusão de dados (Chays et al. 2000). Nesse contexto, existem vários estudos sendo conduzidos para o teste dessas aplicações. Esses estudos envolvem geração de dados, teste do projeto e da aplicação de base de dados (Chan e Cheung 1999, Chays et al. 2000, Chays e Deng 2003, Deng et al. 2005, Kapfhammer e Soffa 2003, Suárez-Cabal e Tuya 2004, Zhang et al. 2001); alguns estudos exploram o uso de informações do esquema para o teste da aplicação de base de dados (Chan et al. 2005, Robbert e Maryanski 1991). Nesse contexto de base de dados relacional, destaca-se o trabalho de Aranha et al. (2000) que aborda o teste de esquema.

Aranha et al. (2000) descrevem a ferramenta *RDBTool* (*Relational Data-Base Testing Tool*), que realiza teste em base de dados relacional para focalizar a estrutura lógica dos dados e os próprios dados. O teste é executado por meio de operações (sentenças SQL) na base de dados para revelar defeitos na definição de atributos e relacionamentos especificados no esquema, empregando critério de teste estrutural. A ferramenta *RDBTool* executa a análise estática do esquema da base de dados em teste, extraindo dados sobre a definição de atributos e relacionamentos e determinando os elementos requeridos com o auxílio do testador, que deve escolher os elementos que serão testados e os critérios de teste que serão empregados. Um exemplo de critério de teste proposto nesse trabalho é o critério "todos os tipos de definição". A partir destas informações são geradas as sentenças SQL para testar cada elemento requerido e executar o teste. O testador analisa os resultados do teste para avaliar a cobertura dos critérios e possíveis defeitos no esquema. A *RDBTool* pode ser aplicada somente no teste de esquemas de base de dados relacional e não gera automaticamente as sentenças SQL.

No contexto de esquemas XML, vários trabalhos abordam o teste de aplicações Web (Di Lucca e Di Penta 2003, Di Lucca et al. 2002a, Di Lucca et al. 2002b, Kung et al. 2000, Liu et al. 2000a, Liu et al. 2000b, Ricca e Tonella 2002); e outros trabalhos testam somente alguns aspectos dessas aplicações; por exemplo, a interação entre componentes Web por meio de mensagens XML (Lee e Offutt 2001) e serviços Web (Offutt e Xu 2004, Xu et al. 2005). Nesse contexto, destacam-se os trabalhos de Li e Muller (2005) e Franzotte e Vergilio (2006) que visam o teste de esquemas.

Li e Muller (2005) propõem operadores de mutação para XML *Schema* que fazem alterações simples no esquema, como troca de um valor ou atributo. Os autores mostram que alguns desses operadores geram modificações que não são descobertas apenas validando os documentos XML, mas não apresentam um processo de teste nem uma ferramenta de aplicação.

Franzotte e Vergilio (2006) introduzem um conjunto de operadores de mutação e aplicam o teste de mutação em esquemas XML com o apoio de uma ferramenta denominada *XTM* (*Tool for XML Schema Testing Based on Mutation*). Os operadores de mutação são subdivididos em operadores de mutação elementar, que alteram valores de atributos e elementos modificando a ordem, o uso e o tipo de dado, por exemplo; e operadores de mutação estrutural, que modificam a estrutura da árvore de um esquema, invertendo, adicionando e removendo nós. Resultados promissores são apresentados, visto ser a técnica baseada em defeitos bastante eficaz em termos de defeitos revelados.

Entretanto, a utilização da *XTM* e de seus operadores é limitada a esquemas XML escritos em XML *Schema*. Além disso, a geração de dados de teste precisa ser feita manualmente pelo testador, que deverá também manualmente identificar esquemas equivalentes.

Ambas as ferramentas (*RDBTool* e *XTM*) só podem ser aplicadas em determinados contextos e não oferecem apoio automatizado à difícil tarefa de geração de dados de teste. Para lidar com essas limitações a ferramenta *XTool* foi implementada e será descrita na próxima seção.

3. Ferramenta *XTool*

A *XTool* (*XML and Relational Database Schema Testing Tool*) apóia o teste de esquemas de dados implementando uma abordagem baseada em defeitos, denominada Análise de Instâncias de Dados Alternativas (AIDA) (Emer et al. 2007a). Essa abordagem tem como objetivo detectar defeitos em esquemas de dados para garantir a integridade dos dados definidos por meio do esquema e manipulados por uma aplicação de software.

A AIDA é genérica, ou seja, pode ser aplicada em esquemas de dados de diferentes contextos, pois emprega um modelo de dados definido por um metamodelo *MM* baseado em MOF (*Meta Object Facility*) (OMG 2006) que representa os esquemas de dados, instanciando e interpretando os componentes do esquema. A Figura 1 apresenta o metamodelo *MM*, que consiste das classes: Elemento (entidades ou objetos), Atributo (características dos elementos), Restrição (restrições associadas aos elementos e atributos) e, da classe associativa: Associação (características referentes à classe Elemento).

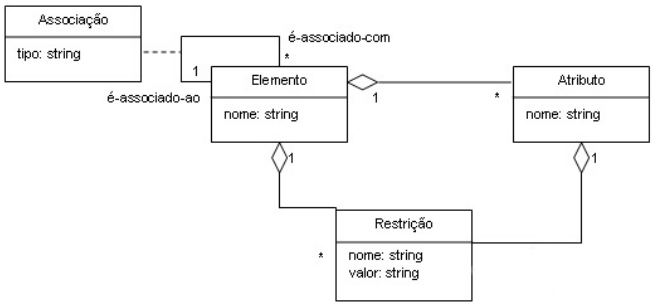


Figura 1. Metamodelo *MM*

Além disso, a AIDA considera defeitos comuns, que podem ser introduzidos no esquema durante o seu desenvolvimento, e a representação formal, que permite a identificação automática de possíveis defeitos. Os defeitos são organizados em classes de defeitos (Tabela 1). As classes de defeitos foram determinadas a partir de inspeções realizadas em esquemas de dados e com base em trabalhos da literatura, tais como: Lee e Offutt (2001) e Offutt e Xu (2004). Um exemplo de classe de defeito relacionada à restrição de domínio é a classe denominada "Valores Enumerados Incorretos", que diz respeito à definição incorreta de um conjunto de valores permitidos para elementos ou atributos. A representação formal providencia a identificação dos elementos, atributos, restrições e relacionamentos entre eles. Desse modo, um esquema de dados é denotado por $S = (E, A, R, P)$, onde:

- E é um conjunto finito de elementos.
- A é um conjunto finito de atributos.

- R é um conjunto finito de restrições referentes ao domínio, definição, relacionamento e conteúdo de elementos ou de atributos.
- P é um conjunto de regras de associação que relacionam elementos, atributos e restrições. Uma regra de P pode ter um dos seguintes formatos. Considere $U = E \cup A$.
  - $p(x, y) \mid x, y \in E \wedge x \neq y;$
  - $p(x, r) \mid x \in E \wedge r \in R;$
  - $p(x, r, SU) \mid x \in U \wedge r \in R \wedge SU = \{u_1, u_2, ..., u_m\} \subset U, \quad \forall u_i \neq x,\ 1 \leq i \leq m, m \geq 1,$ onde m é o número de elementos e atributos de SU.

Tabela 1. Restrições e classes de defeitos

Classe de defeito	Descrição
Grupo 1 - Restrições de domínio	
Tipo de dado incorreto	Definição incorreta de tipo de dado de um elemento
Valor incorreto	Valor padrão ou fixo definido incorretamente para o elemento
Valores enumerados incorretos	Definição incorreta do conjunto de valores permitidos para o elemento
Valores máximos e mínimos incorretos	Definição incorreta de valores limites definidos para o elemento
Tamanho incorreto	Número de caracteres máximos e mínimos permitidos para um elemento definidos incorretamente
Limite de dígitos incorreto	Número de dígitos máximos e mínimos permitidos para um elemento definidos incorretamente
Modelo de valores incorreto	Definição incorreta da seqüência de valores permitidos para um elemento
Tratamento de espaço em branco incorreto	Definição incorreta do tratamento para espaço em branco
Grupo 2 - Restrições de definição	
Uso incorreto	Definição incorreta de um elemento como opcional ou obrigatório
Unicidade incorreta	Definição incorreta do número de ocorrências permitidas para um elemento
Identificador incorreto	Definição incorreta de uso e unicidade para elementos que compõem o identificador de um elemento complexo
Grupo 3 - Restrições de relacionamento	
Ocorrência incorreta	Definição incorreta do número máximo e mínimo de repetições de um elemento
Ordem incorreta	Definição incorreta da ordem de ocorrência para os elementos-filho de um determinado elemento
Associação Incorreta	Definição incorreta de uma associação: cardinalidade, generalização/especialização, agregação, elemento associativo
Grupo 4 - Restrições semânticas	
Condição incorreta	Definição incorreta de restrições do conteúdo de um elemento em relação ao conteúdo de outro elemento

3.1 Funcionalidades da *XTool*

A Figura 2 ilustra as principais funcionalidades da ferramenta *XTool* que são descritas a seguir. Na Figura 2 pode ser observado que o teste é iniciado pelo testador, que disponibiliza o esquema em teste e uma instância de dado associada a esse esquema para a ferramenta.

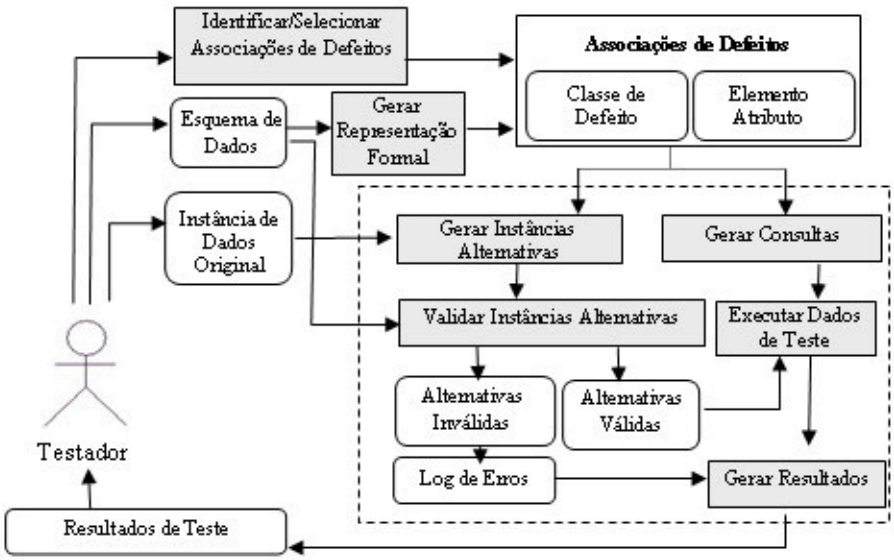


Figura 2. Funcionalidades da *XTool*

A ferramenta gera a representação formal para o esquema. Essa representação formal é utilizada para estabelecer automaticamente associações de defeitos presentes no esquema. Uma associação de defeito é um elemento (entidade) ou atributo e restrições aos dados do esquema associadas a uma classe de defeito. Após a geração das associações de defeitos, elas podem ser selecionadas pelo testador, que poderá também introduzir outras associações de defeitos, utilizando uma caixa de diálogo da ferramenta, se alguma outra associação puder ser estabelecida. Essas outras associações, inseridas pelo testador, podem representar restrições aos dados que podem estar ausentes no esquema, mas que o testador gostaria também de testar.

As associações de defeitos selecionadas norteiam a geração das instâncias de dados alternativas que podem ser documentos XML ou instâncias de base de dados relacional, de acordo com o esquema em teste. As instâncias de dados alternativas são geradas automaticamente por pequenas alterações nos dados da instância de dados original (disponibilizada pelo testador). As instâncias de dados alternativas representam os defeitos descritos pelas classes de defeitos. Essas instâncias passam por um processo de validação e são separadas em válidas e inválidas com relação às definições do esquema em teste.

As consultas também são geradas automaticamente, de acordo com as associações de defeitos selecionadas, e utilizam padrões de consulta previamente estabelecidos em uma linguagem de consulta adequada ao tipo de dado associado ao esquema em teste.

Na abordagem implementada pela *XTool*, os dados de teste são formados por uma instância de dados alternativa válida e por uma consulta que deve ser executada nessa instância de dados alternativa, de acordo com uma associação de defeito. O resultado dessa consulta deve ser avaliado pelo testador e pode indicar um defeito no esquema em teste.

As instâncias de dados alternativas inválidas também são resultados de teste e podem ser analisadas pelo testador. O testador verifica se os resultados obtidos com o teste estão de acordo com os resultados esperados, sempre que esses resultados diferem, um defeito foi revelado.

É interessante observar que o teste do esquema é realizado independentemente da aplicação que utiliza os esquemas, porém, é necessário que a especificação dos dados da aplicação esteja disponível para que os resultados de teste sejam analisados pelo testador. Nada impede que as instâncias de dados alternativas geradas possam também ser utilizadas em uma outra etapa como dados de teste para a aplicação.

3.2 Aspectos de Implementação

A *XTool* realiza atualmente o teste de esquemas XML escritos em XML *Schema* e o teste de esquemas de base de dados relacionais para o SGBD *PostGreSQL* (PostGre 2006).

As funcionalidades da *XTool* foram implementadas de acordo com o diagrama de classes apresentado na Figura 3. Essas classes são brevemente descritas a seguir:

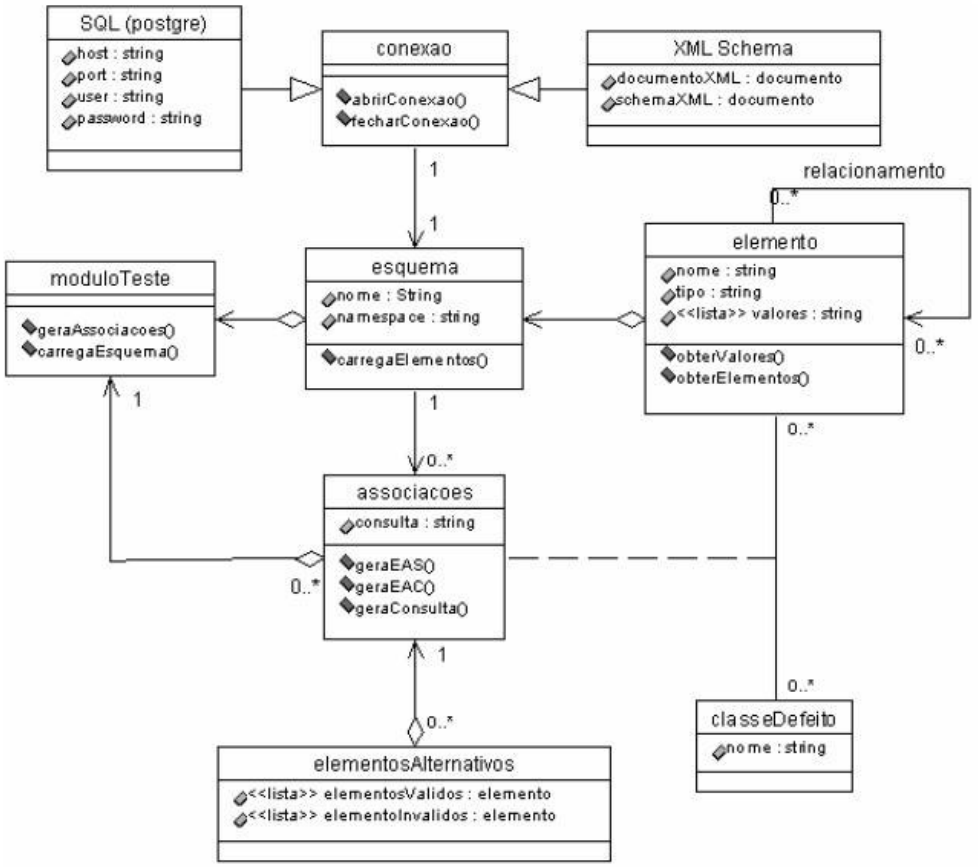


Figura 3. Diagrama de classes da XTool

- moduloTeste → classe principal, responsável pela interação entre a interface do usuário e as demais classes;
- esquema → classe responsável por armazenar o esquema em teste e algumas outras informações relevantes sobre o esquema;
- elemento →classe que representa os elementos de um esquema;
- classeDefeito → classe que representa as classes de defeitos previamente estabelecidas;
- associações → classe que representa as associações de defeitos;

- elementosAlternativos → classe responsável por armazenar as instâncias de dados alternativas;
- conexao → classe responsável por gerenciar a conexão com o esquema. Essa classe atualmente contém duas subclasses que permitem o teste de esquemas *PostGreSQL* e XML *Schema*. Para configurar a ferramenta para o teste de um outro tipo de esquema, uma nova sub-classe precisa ser criada.
    - conexao SQL (postgre) → subclasse da classe conexão, responsável pela conexão com uma base de dados *PostGreSQL*;
    - conexao XMLSchema → subclasse da classe conexão, responsável pelo acesso a documentos XML.

A *XTool* foi implementada na linguagem Java com o emprego do DOM (*Document Object Model*) (W3C 2006c) para manipular e validar documentos XML; Qexo (Bothner 2006) para consultar os documentos XML por meio da linguagem *XQuery* (W3C 2006d); JDBC (*Java Database Connectivity*) (SUN 2006) para manipular e consultar informações em bases de dados *PostGreSQL* (PostGre 2006) por meio das linguagens DQL (Linguagem de Consulta de Dados) e DML (Linguagem de Manipulação de Dados) que são subconjuntos da SQL (*Structured Query Language*).

4. Testando um Esquema com a *XTool*

Para ilustrar o procedimento de uso da ferramenta *XTool*, bem como as informações por ela produzidas, essa seção mostra como é realizado o teste de um esquema XML. Para isso, considere o fragmento do esquema de uma loja de cd's a ser testado e, o fragmento da instância de dados original (documento XML original, fornecido pelo usuário) apresentado na Figura 4. O esquema contém a definição dos dados referentes a cd's de música que podem ser armazenados em documentos XML associados a esse esquema.

```
<schema>
 <element
name="cds"><complexType><sequence>
 <element name="cd" maxOccurs="unbounded">
 <complexType> <sequence>
 <element name="titulo" type="string"
minOccurs="1"  maxOccurs="1" />
 <element name="artista" type="string"
minOccurs="1"  maxOccurs="unbounded" />
…
 <element name="ano" type="integer"
minOccurs="1"  maxOccurs="1"/>
…
 <attribute name="duracao" type="string"
use="optional"/>
 </complexType> </element>
 </sequence> </complexType> </element>
</schema>
```

(a) Exemplo 1: Fragmento do esquema XML para dados sobre cd's disponíveis em uma loja de música

```
<?xml version="1.0" ?>
<cds
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
  xsi:schemaLocation="cd.xsd">
  <cd genero="instrumental" duracao="00:46:32">
    <titulo>Sucessos do Cinema</titulo>
    <artista>Richard Clayderman</artista>
    <musica>I just called to say I love
      you</musica>
    <musica>I will always love you</musica>
    <musica>Unchained melody</musica>
    <musica>Theme from love story</musica>
    <musica>How deep is your love</musica>
    <gravadora>Polygram</gravadora>
    <ano>1996</ano>
  </cd>
...
</cds>
```

(b) Fragmento da instância de dados original

Figura 4: Fragmento de um esquema e de um documento XML associado ao esquema

A Figura 5 apresenta um fragmento da representação formal gerada pela *XTool* para descrever os elementos, atributos, restrições e regras de associação entre elementos, atributos e restrições do esquema em teste apresentado na Figura 4.

E = {cds, cd, titulo, artista, musica, gravadora, ano}
A = {genero, duracao}
R = {ordem, ocorrência, tipo, uso}
P = {p_1(cds, cd), p_2(cds, ordem, cd),
p_3(cd, titulo), p_4(cd, artista),
p_5(cd, musica), p_6(cd, gravadora),
p_7(cd, ano), p_8(cd, genero),
p_9(cd, duracao), p_{10}(cd, ordem, titulo,
artista, musica, gravadora, ano, genero,
duracao), p_{11}(cd, ocorrência),
p_{12}(titulo, ocorrência), p_{13}(titulo, tipo)
p_{14}(artista, ocorrência), ...
p_{20}(ano, tipo), ...
p_{24}(duração, uso), p_{25}(duracao, tipo)}

Figura 5: Fragmento de representação formal

No Exemplo 1 é ilustrada uma associação de defeito identificada na representação formal do esquema XML. Essa associação de defeito relaciona o elemento "ano" à classe de defeito "Tipo de Dado Incorreto" por meio da restrição "tipo" (*type*) definida para o elemento "ano".

Exemplo 1: Exemplo de associação de defeito identificada no esquema XML em teste
Elemento : ano
Classe de Defeito : Tipo de dado Incorreto
Restrição : tipo

O Exemplo 2 ilustra o conteúdo do elemento "ano" na instância de dados original associada ao esquema do Exemplo 1

Exemplo 2: Exemplo de conteúdo do elemento "ano"
<cds><cd> ... <ano>1996</ano>....</cd></cds>

Com base na associação de defeito do Exemplo 1 e das outras associações identificadas no esquema XML, as instâncias de dados alternativas são criadas por meio de uma modificação simples na instância de dados original (documento XML). Exemplos de alterações que podem ser feitas na instância de dados original de acordo com associação descrita anteriormente podem ser vistos no Exemplo 3. Cada uma dessas alterações gera uma instância de dados alternativa ou um documento XML alternativo.

Exemplo 3: Exemplos de possíveis alterações no conteúdo do elemento "ano"
<cds>...<ano>1996</ano>....</cds>
<cds>...<ano>abc1234</ano>....</cds>
<cds>...<ano>-1234</ano>....</cds>
<cds>...<ano>1234.56</ano>....</cds>

As instâncias de dados alternativas são validadas em relação ao esquema XML em teste e separadas em válidas e inválidas. No Exemplo 4, as alterações propostas no Exemplo 3 são classificadas em válidas ou inválidas.

Exemplo 4: Alterações válidas ou inválidas
Alterações válidas

```
<cds>...<ano>1996</ano>....</cds>
<cds>...<ano>-1234</ano>....</cds>
```

Alterações inválidas

```
<cds>...<ano>abc1234</ano>....</cds>
<cds>...<ano>1234.56</ano>....</cds>
```

As associações também guiam a geração de consultas em *XQuery*. Essas consultas e os documentos XML alternativos válidos formam dados de teste. Os dados de teste são executados e o resultado dessa execução é avaliado pelo testador de acordo com a especificação. Por exemplo, a associação de defeito do Exemplo 1 gera a consulta ilustrada no Exemplo 5. Essa consulta retorna o conteúdo de cada elemento "ano" do documento XML alternativo válido que está sendo consultado.

Exemplo 5: Consulta para a associação de defeito do Exemplo 1

```
let $doc := document("cd.xml")
for $i in $doc\\cds\cd\ano
return <resultado>{$i}</resultado>
```

O resultado obtido após a execução do dado de teste formado pela consulta do Exemplo 5 e pelos documentos XML alternativos válidos, gerados por meio da mesma associação de defeito que produziu essa consulta, é apresentado no Exemplo 6.

Exemplo 6: Resultado da consulta do Exemplo 5

```
<resultado>
  <ano>1996</ano>
  <ano>-1234</ano>
<resultado>
```

O resultado obtido é comparado com o resultado esperado. O conteúdo "-1234" obtido da consulta ao elemento "ano" é um valor numérico inválido para o conteúdo de um elemento que representa "ano". Dessa forma, pode ser concluído que o esquema XML sob teste permitiu que um documento XML com um dado incorreto fosse considerado válido. Portanto, o esquema possui um defeito na definição do elemento "ano". Esse defeito pode ser revelado porque é coberto pela classe de defeito "Restrições de Domínio - Tipo de Dado Incorreto".

5. Resultados de Estudos de Caso

A fim de validar a implementação da ferramenta *XTool* e a aplicabilidade da abordagem, foram efetuados estudos de caso para realizar o teste de esquemas XML e de esquema de base de dados relacional. O objetivo foi verificar o custo, em termos do número de dados de teste (consultas e associações) necessários, e a eficácia, em termos do número de defeitos revelados. Para realizar essa análise, os defeitos foram revelados durante o teste da aplicação em desenvolvimento, portanto os defeitos são naturais e não semeados.

Os resultados obtidos em ambos os estudos de caso foram bastante animadores e são resumidos a seguir. No contexto de XML, os esquemas utilizados definem dados de um sistema de matrícula e de um sistema de biblioteca. Esses esquemas contêm dados de estudantes ("aluno.xsd"); disciplinas disponíveis no curso ("disciplina.xsd"); usuários registrados na biblioteca ("usuario.xsd"); e títulos disponíveis na coleção da

biblioteca (“obras.xsd”). No contexto de base de dados relacional, o esquema define dados referentes ao histórico de alunos egressos de uma Universidade.

Na Tabela 2 são apresentadas características dos esquemas, como: número de elementos ou entidades e atributos; e número de registros armazenados na instância de dados original do esquema correspondente.

Tabela 2. Características dos esquemas

Tipo de esquema	Esquema	Elementos/Entidades	Atributos	Registros
XML	Aluno	10	0	5
	Disciplina	10	0	6
	Obras	11	0	7
	Usuário	8	0	6
Base de dados relacional	Universidade	19	73	126

A Tabela 3 apresenta os resultados de teste da *XTool* para cada esquema. Nessa tabela, o número de associações de defeitos identificadas e selecionadas; o número de instâncias alternativas geradas; o número de consultas produzidas pela *XTool*; e, o número de defeitos revelados no teste de cada esquema são apresentados. Nessa tabela pode ser observado, por exemplo, que a partir das 16 associações de defeitos identificadas para o esquema “aluno.xsd” foram revelados 4 defeitos no esquema.

Tabela 3. Resultados de teste da *XTool*

Tipo de esquema	Esquema	Associações de defeitos	Alternativas válidas/inválidas	Consultas geradas	Defeitos revelados
XML	Aluno	16	241/42	16	4
	Disciplina	16	283/43	16	4
	Obras	16	252/138	16	2
	Usuário	12	187/73	12	3
Base de dados relacional	universidade	297	942/1720	1240	27

Nestes estudos de caso, a classe de defeito mais eficaz nos esquemas XML foi a classe de defeito “Restrições de Domínio – Tipo de Dado Incorreto” e no esquema da base de dados relacional foi a classe de defeito “Restrições de Definição – Uso Incorreto”. Além dessas classes de defeitos, as classes “Restrições de Relacionamento – Ocorrência Incorreta”; “Restrições de Domínio – Tamanho Incorreto”; “Restrições de Domínio – Dígito Incorreto”; “Restrições de Domínio – Valores Enumerados Incorretos”; e “Restrições de Relacionamento – Associação Incorreta” também revelaram defeitos nos esquemas testados.

Os estudos de caso mostraram que a ferramenta apóia a aplicação da AIDA, auxiliando para a detecção de defeitos em esquemas XML e esquemas de base de dados relacional. Além disso, os resultados obtidos com os estudos de caso permitem afirmar que o maior custo da aplicação da abordagem com o apoio da *XTool* está na análise dos resultados de teste; dado que, essa análise é feita manualmente e envolve uma grande quantidade de dados devido ao número de instâncias de dados alternativas e de consultas geradas no processo de teste. No entanto, essa é uma limitação inerente à atividade de teste relativa à automatização do oráculo.

Além desses estudos de caso foram realizados outros dois, nos quais os defeitos foram semeados nos esquemas em teste, gerando esquemas mutantes. Um dos estudos

de caso utilizou a ferramenta *XTM* para gerar diferentes esquemas XML mutantes incorretos (Emer et al. 2007b). Esses esquemas foram testados com a *XTool* e todos tiveram seus defeitos revelados, mostrando que as classes de defeitos da *XTool* cobrem os defeitos descritos pelos operadores de mutação da *XTM*. O outro estudo de caso foi realizado por meio da inserção ad hoc de defeitos em esquemas de base de dados relacional gerando esquemas mutantes (Emer et al. 2007c). Esses esquemas mutantes foram testados pela *XTool* e todos os defeitos inseridos foram cobertos pelas classes de defeitos da *XTool* e, portanto, revelados. É importante salientar que os defeitos semeados foram considerados revelados com o uso da *XTool*, se os resultados de teste dos mutantes fossem diferentes dos resultados de teste do esquema original correspondente.

6. Conclusões e Trabalhos Futuros

Este artigo apresentou a ferramenta *XTool* que apóia a abordagem de teste denominada AIDA. Essa abordagem de teste é baseada em classes de defeitos que descrevem possíveis defeitos inseridos durante a definição ou evolução de um esquema de dados. Dessa forma, essa abordagem de teste pode revelar os defeitos cobertos pelas classes de defeitos e que estejam relacionados à definição incorreta ou ausente de restrições aos dados no esquema. Diferentemente das abordagens de teste implementadas pelas ferramentas encontradas na literatura (*RDBTool e XTM*), a abordagem de teste implementada é genérica, podendo ser aplicada em diferentes tipos de esquemas. Atualmente, a ferramenta *XTool* está configurada para o teste de dois tipos de esquemas: *XML Schema* e *PostGreSQL Schema*, utilizados respectivamente para definir dados em formato XML e dados de base de dados relacional.

A *XTool* foi empregada em estudos de caso, nos quais esquemas XML e um esquema de base de dados relacional foram testados. Esses estudos de caso mostraram que o uso da *XTool* é imprescindível para apoiar a aplicação da abordagem de teste AIDA, possibilitando a descoberta de defeitos, em esquemas de dados, cobertos pelas classes de defeito definidas. Os estudos de caso também permitiram a observação da necessidade de refinar a *XTool*, por exemplo, implementando um suporte ao oráculo para facilitar a comparação dos resultados obtidos com o teste com os resultados esperados, mesmo que não seja possível automatizar completamente essa tarefa. A grande vantagem da ferramenta comparada com as demais é que a tarefa de geração de dados de teste (consultas e instâncias de dados) é realizada automaticamente, sem a participação do usuário. Isso contribui para reduzir o esforço de teste. Além disso, o teste do esquema pode ser efetuado independentemente da disponibilidade da aplicação. As instâncias de dados alternativas geradas podem ser, em um segundo momento, utilizadas como dados de teste para o teste das aplicações. Isso deverá ser avaliado em trabalhos futuros.

A *XTool* pode ser estendida para testar esquemas em outros contextos, tais como: base de dados XML e serviços Web. Novos estudos de caso devem ser realizados para explorar outros esquemas e contextos. Os resultados desses novos estudos de caso poderão contribuir para o refinamento das classes de defeitos definidas na abordagem implementada.

Referências

Aranha, M. C. L. F. M.; Mendes, N.C.; Jino, M.; Toledo, C.M.T. (2000) "RDBTool: A Support Tool for Testing Relational Database". XI ICST, Int. Conference of Software Technology.

Bothner, Per. (2006) “Qexo: The GNU Kawa implementation of XQuery”, http://www.gnu.org/software/qexo/, 2005. Acessado em 2006.

Chan, M.; Cheung, S.. (1999) “Testing Database Applications with SQL Semantics”. In Proceedings of the 2nd International Symposium on Cooperative Database Systems for Advanced Applications, pp 364-375, March.

Chan, W. K.; Cheung, S. C.; Tse, T. H.. (2005) “Fault-Based Testing of Database Application Programs with Conceptual Data Model”. In Proceedings of the Fifth International Conference on Quality Software, pp 187-196.

Chays, D.; Deng, Yuetang. (2003) “Demonstration of AGENDA Tool Set for Testing Relational Database Applications”. In Proceedings of the 25th International Software Engineering Conference. IEEE Computer Society, pp 802 – 803. May.

Chays, David; Dan, Saikat; Frankl, Phyllis G.; Vokolos, Filippos I.; Weyuker, Elaine J.. (2000) “A Framework for Testing Database Applications”. In Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, Vol. 25 Issue 5, August.

Chen, P. P.. (1976) “The Entity-Relationship Model – Toward a Unified View of Data”. ACM Transactions on Database Systems (TODS), Vol. 1, No 1, pp 9-36.

Deng, Yuetang; Frankl, Phyllis; CHAYS, David. (2005) “Testing Database Transactions with AGENDA”. In Proceedings of the 27th International Conference on Software Engineering. ACM Press, May.

Di Lucca, G.A. e Di Penta, M.. (2003) “Considering Browser Interaction in Web Application Testing”. In Proceedings of the 5th IEEE Intl. Workshop on Web Site Evolution. IEEE Computer Society Press.

Di Lucca, G.A.; Fasolino, A.R.; Faralli, F. e De Carlini, U.. (2002a) “Testing Web applications”. In Proceedings of the International Conference on Software Maintenance, pages 3–6. IEEE Press, October.

Di Lucca, G.A.; Fasolino, A.R.; Pace, F.; Tramontana, P. e De Carlini, U.. (2002b) “WARE: A tool for the Reverse Engineering of Web Applications”. In Proceedings of the VI Eur. Conf. on Software Maintenance and Reengineering. IEEE Computer Society Press, March.

Emer, M. C. F. P.; Vergilio, S. R.; Jino, M. (2007a) “A Fault-Based Testing of Data Schemas”. In Proc. of the 19th Intl. Conference on Software Engineering and Knowledge Engineering (SEKE 2007), July.

Emer, M. C. F. P.; Nazar, I. F.; Vergilio, S. R.; Jino, M. (2007b) “Evaluating a Fault-Based Testing Approach for XML Schemas”. VIII Latin-American Test Workshop. Cuzco, Peru, March.

Emer, M. C. F. P.; Vergilio, S. R.; Jino, M.; Nazar, I. F.; Caxeiro, P. V. (2007c) “Uma Avaliação do Teste Baseado em Defeitos em Esquemas de Banco de Dados Relacional”. In Proc. of the 1st Brazilian Workshop on Systematic and Automated Software Testing (SAST 2007) em conjunto com o XXI Simpósio Brasileiro de Engenharia de Software (SBES 2007), outubro.

Franzotte, L; Vergilio, S.R. (2006) “Applying Mutation Testing to XML Schemas”. In Proc. of the 18th Inl. Conference on Software Engineering and Knowledge Engineering (SEKE2006).

Kapfhammer, Gregory M.; Soffa, Mary Lou. (2003) “A Family of Test Adequacy Criteria for Database-driven Applications”. In Proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering ESEC/FSE-11, Volume 28 Issue 5. ACM Press, September.

Kung, D.C.;. Liu, C.H e Hsia, P.. (2000) “An Object-Oriented Web Test Model for Testing Web Applications”. In Proceedings of the 24th Annual International Computer Software and Applications Conference, COMPSAC 2000, pages 537–542.

Lee, S.C. e Offutt, J.. (2001) “Generating test cases for XML-based web component interactions using mutation analysis”. In Proceedings of the 12th International Symposium on Software Reliability Engineering, pages 200–209. IEEE Press, November.

Li, J. B.; Miller, J. (2005) “Testing the Semantics of W3C XML Schema”. In Proc. of the 29th Annual Intl. Computer Software and Applications Conference (COMPSAC-2005), Julho.

Liu, C.H.; Kung, D.C. e Hsia, P.. (2000a) “Object-based Data Flow Testing of Web Applications”. In Proceedings of the 1st Asia-Pacific Conference on Quality Software, pages 7–16. IEEE Press.

Liu, C.H.; Kung, D.C.; Hsia, P. e Hsu, C.T.. (2000b) “Structural Testing of Web Applications”. In Proceedings of the 11th International Symposium on Software Reliability Engineering, pages 84–96. IEEE Press.

Offutt, J.; Xu, W. (2004) “Generating Test Cases for Web Services Using Data Perturbation”. In Proceedings of the TAV-WEB, volume 29. ACM SIGSOFT SEN, September.

OMG (2006). “Meta-Object Facility Core Specification Version 2.0”. http://www.omg.org/ cgi-bin/doc?formal/ 2006- 01-01, January 2006. (acessado em 2006).

PostGre (2006) “PostGreSQL”, http://www.postgresql.org/docs/, Acessado em 2006.

Ricca, F. e Tonella, P.. (2002) “Analysis and Testing of Web Applications”. In Proceedings of the 23rd International Conference on Software Maintenance, pages 25–34. IEEE Press, May.

Robbert, M. A.; Maryanski, F. J.. (1991) “Automated Test Plan Generator for Database Application Systems”. In Proceedings of the ACM SIGSAMLL/PC Symposium on Small Systems, pp 100-106.

Suárez-Cabal, M. J.; Tuya, J.. (2004) “Using a SQL Coverage Measurement for Testing Database Applications”. In Proceedings of the 12th International Symposium on the Foundations of Engineering. November.

SUN (2006) “Java Database Connectivity (JDBC)”, http://java.sun.com /javase/technologies/database/, Acessado em 2006.

W3C (2006a) “Extensible Markup Language (XML)”, http://www.w3c.org/XML/, Acessado em 2006.

W3C (2006b) “XML Schema”, http://www.w3c.org/XML/Schema/, Acessado em 2006.

W3C (2006c) “Document Object Model (DOM)”, http://www.w3c.org/DOM/, Acessado em 2006.

W3C (2006d) “XML Query Language. (XQuery)”, http://www.w3c.org/XML/Query/. Acessado em 2006.

Xu, W.; Offutt, J. e Luo, J.. (2005) “Testing Web Services by XML Pertubation”. In Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering. IEEE.

Zhang, Jian; Xu, Chen; Cheung, S.-C.. (2001) “Automatic Generation of Database Instances for White-box Testing”. In Proceedings of the 25th Annual International Computer Software and Applications Conference, 2001, pp 161 – 165, October.

Security Assessment and Testing Tools Information Repository

Naaliel Mendes[1], João Durães[1], Marco Vieira[1], Henrique Madeira[1]

[1] CISUC, Department of Informatics Engineering – University of Coimbra
3030-290 Coimbra, Portugal

{naaliel,jduraes,mvieira,henrique}@dei.uc.pt

Abstract. *This paper presents a repository for storing and comparing the characteristics of security assessment and testing tools. This repository supports information on the tools from different sources, ranging from specialized web sites and magazines to academic publications. This is in fact a powerful mean to share and disseminate information on security testing tools and to raise security issues awareness. It can also represent a very important information source for researchers and practitioners interested in exploring those tools. A working version of the repository is already online.*

1. Introduction

Security assessment and testing tools such as vulnerability scanners and penetration testing tools have become a key factor to evaluate the security configuration of computer systems and applications. These tools aim to find and to prevent system vulnerabilities that could be discovered and exploited by attackers. In addition, the simulation of many sorts of attacks (such as denial of service, cross-site scripting, SQL injection, and buffer overflow) using these tools is aimed at helping system administrators to improve the security configuration of their systems in order to protect them against known attacks. In practice, the use of these tools ranges from end-users to academia and industry.

One important problem is the selection of the right tool for a given concrete scenario, as there is a variety of tools and their features are not easy to compare. A logical solution is to build an information repository to store relevant information on existing tools and help users on their decision when selecting a testing tool.

A repository is simply an infrastructure aimed at collecting data and offering functionalities to help user on finding, analyzing, comparing, and disseminating the repository data. A repository addressing security tools should store characteristics such as target system, type of addressed attack, and type of addressed vulnerability. Additionally, characteristics not directly related to the tools such as academic publications reporting on the utilization of tools are also relevant and should be stored in the repository.

With the prevalence of internet, repositories based on web-enabled tools have increased significantly the power of dissemination and sharing of repository data. Taking into account this web dissemination ability, many repository initiatives have been proposed on the web in several research topics (examples are given in the next section), also supporting the exploration of the stored data through business intelligence

techniques [1]. However, security assessment and testing tools have not been addressed by these previous initiatives.

In this paper we present the Security Assessment and Testing Tools Information Repository (SATTIR). SATTIR provides functionalities ranging from the submission of information on tools to the analysis of such information using well-known techniques of data exploration such as multidimensional analysis. To the best of our knowledge, this is the first repository in this research topic that addresses all the following characteristics:

- **Indexation of tools information from different sources.** This aims to collect and to index the information about tools, supporting contributions provided by repository users. This solves the problem of user's dependence on search engines and reading of publications to find out the state-of-the-art of security assessment and testing tools.
- **Comparison of tools.** The repository allows a detailed comparison of the tools characteristics and helps solving the problem of having to install and test the tools to assess their characteristics in detail. Today, the existence of multiple information sources on tools available (e.g., conferences, magazines, and announces of web sites) and the variety of the information formats have considerably increased the difficulties in comparing tools. This is a key contribution for end-users and system administrators, as they will be able to compare quickly available tools in order to find out which ones fit exactly their security testing purposes.
- **Support for advanced information analysis.** Technologies for information analysis such as Information Retrieval and On-line Analytical Processing (OLAP) are part of the repository functionalities. Data retrieval technologies explore information represented by text data. OLAP tools support the analysis of multidimensional data (data warehousing), being traditionally used to decision support analysis. The means for multidimensional analysis in SATTIR allow powerful comparisons of the tool characteristics, being possible, for instance, to query how many times a given tool was cited in a given type of publication (e.g., academic papers, magazines). Obviously, the significance of the conclusions taken from multidimensional analysis relies on factors as the diversity and amount of information stored in SATTIR. The information quality (a classical problem in OLAP systems) is not an issue given that we have defined procedures to assure that only real tools are taking part of the repository.
- **Dissemination and information sharing.** The repository is made universally available by web-enabled tools, allowing the sharing of the information on tools to end-users and to the industry and academic communities.

The structure of this paper is as follows. The next section presents related work. Section 3 shows the architecture and the underlying technology of SATTIR. The data storage infrastructure of SATTIR is detailed in Section 4. Section 5 discusses on the potential users and SATTIR exploitation scenarios. Section 6 presents the overview of SATTIR functionalities. Section 7 presents the conclusion and ongoing work.

2. Related Work

The collection and storing of data for future analysis or historical purposes continues being a hot topic in our days. A quick search for repositories on the web shows the existence of hundreds of repositories as well as the existence of software packages for building such repositories. These repositories cover different subject areas, being sponsored by universities, industries, and governments. This section presents some of these initiatives within the context of computer science. Sub-section 2.1 presents widely used repositories platforms that are employed for building new repositories and also shows examples of institutional repositories built on those platforms. Subsections 2.2 and 2.3 present respectively examples of data and software repositories. The subsection 2.3 is the one that presents the repository initiatives related directly to security assessment and testing tools.

2.1. Repositories Platforms

The popularity of repositories has motivated the proposal of platforms to help users in the creation of new repositories. The main idea is to endow community with tools that mitigate the technical difficulties of implementing a new repository (coding of scripts and web site interface, backups scripts, etc), which in fact demand considerable time and hard work from programmers and administrators. These tools typically offer an infrastructure where users may store data in a typical relational database, submit and search data through a web catalog interface.

Repository platforms have been mainly addressed in studies that demand the storage of huge amount of data, such as libraries, museums, and universities (called as institutional repositories). To help administrators in choosing platforms that better fit to theirs institution purposes, the Open Society Institute compared the characteristics of the most relevant open-source institutional repository platforms [2][3]. Here we present only three of the most representative of these platforms.

Eprints is a platform for building repositories, providing an easy and fast way "to set up repositories of research literature, scientific data, student theses, project reports, multimedia artifacts, teaching materials, scholarly collections, digitized records, exhibitions and performances" [4]. This platform runs over the MySQL database and the Apache Web Server and was coded in Perl programming language. One of the advantages of Eprints is that institutions can get it up and running relatively quickly and with a minimum of technical expertise, due to the size of the installed base. The archive section of the Eprints web site presents hundreds of repositories that are using this platform. Examples of repositories using Eprints are the Archive of European Integration (http://aei.pitt.edu/), the Cambridge University Engineering Department Publications Database (http://publications.eng.cam.ac.uk/), and Indian Institute of Science (http://eprints.iisc.ernet.in/).

DSpace is an open source dynamic digital repository solution for accessing, managing and preserving scholarly works [5]. In fact, this platform is also focused on the problem of long-term preservation of deposited research material. This platform runs over the PostgresSQL or Oracle databases and was coded in Java programming language. To search information in text files, DSpace use the Lucene search engine [6]. One of the main strengths of DSpace is the capacity of recognizing and managing a

large number of file format and mime types. Some of the most common formats managed within the DSpace environment are PDF and Word documents, JPEG, MPEG, TIFF files. DSpace argues that over two hundred academic and cultural institutions use their platform. Examples of repositories using DSpace are The Hong Kong University of Science and Technology Library (http://repository.ust.hk/dspace/), The American Museum of Natural History (http://digitallibrary.amnh.org/dspace/), and The Royal Naval Museum (http://www.seayourhistory.org.uk/).

The **Flexible Extensible Digital Object Repository Architecture** (Fedora) is a repository architecture for digital libraries proposed by Cornell University [7]. The implementation of this architecture is currently available through the Fedora software that is promoted by Fedora Commons organization [8]. It is a non-profit organization aimed at providing "sustainable technologies to create, manage, publish, share and preserve digital content as a basis for intellectual, organization, scientific and cultural heritage". This platform runs over MySQL, McKoi, and Oracle database and Tomcat web server, and was coded in Java programming language. Fedora's features cover aspects such as service-oriented architecture (repository access and management via web services), content versioning (repositories can maintain a record of how digital objects have changed over time), and basic search in the repository. Fedora tool package is available in Fedora web site, allowing the contribution of users with the proposal of new Fedora services. Examples of digital libraries repositories using Fedora are Encyclopedia of Chicago (http://www.encyclopedia.chicagohistory.org/), The National Science Digital Library (http://nsdl.org/), and The University of Virginia Library (http://www.lib.virginia.edu/digital/).

Although existing repository platforms are of paramount importance to the field of digital libraries, and even considering other initiatives [9] [10], [11], there is no repository platform addressing needs of the field of security assessment and testing tools. However, the study of the presented repositories was essential to assess the state-of-the-art of the top technologies employed in the construction of those repositories.

2.2. Data Repositories

Data repositories are an excellent resource to store and share information for research purposes. One type of valuable information that can be shared through data repositories is the results from field data studies. This subsection describes some of the fault and failure data repositories currently available.

The **Data & Analysis Center for Software** (DACS) is a Department of Defense Information Center supporting research on software reliability and quality. It serves as centralized source for data related to software metrics. The DACS maintains the Software Life Cycle Experience Database (SLED). This repository is intent to support the improvement of the software development process. The SLED is organized into nine data sets covering all phases and aspects of the software lifecycle [12][13]. Examples of these datasets are the DACS Productivity Dataset [14], the NASA/SEL Dataset, and the Software Reliability Dataset [15].

The **Metrics Data Program** (MDP) Repository is a database maintained by the NASA Independent Verification and Validation facility [16]. The repository is aimed at the dissemination of nonspecific data to the software community and it is made

available to the general public at no cost. All the data available in the repository is sanitized by the project representatives, and all the necessary clearances are provided. Users of the repository are free to analyze the data for their specific research goal. The repository contains data on the software projects that were collected and validated by the MDP program, spanning more than 8 years and including more than 2700 error reports. The information stored in the repository consists of error data and software metrics and error data at the function/method level.

The **Software Reference Fault and Failure Data Project** [17] is maintained by the National Institute of Standards & Technology and is aimed at the development of metrology, taxonomy and repository for reference data for software assurance. The project maintains a repository on software fault data specifically aimed at to help industry protect against releasing software systems with faults and to help assess software system quality by providing statistical methods and tools for analysis of software systems. The repository is available to the public upon request. The access to the information online allows users to view data and execute simple queries. Analytical and statistical use of the data is possible through a program developed within the project and available to the public.

The **Computer Failure Data Repository** (CFDR) is a public repository on computer failure data [18][19] supported by USENIX. The repository is aimed at the acceleration of the research on system reliability with the ultimate goal of reducing or avoiding downtime in computer systems. The CDFR repository is open to both obtaining and contributing data kind of users. The repository comprises nine independent datasets focusing mainly on very large storage systems. Example datasets are: the LANL dataset which covers 22 high performance computer systems encompassing more than 4700 machines during 9 years, and the HPC3 dataset which covers a 256 node cluster with 520 drives. The repository information covers many aspects, including: software failures, hardware failures, operator errors, network failures, and operational environment problems. Each failure is described by the time of start and end, the node affected and the classification of root cause. The raw data is available to the public [9] through a web interface. To-date the project does not offer online capability for analytic and statistical data-processing.

The **AMBER Data Repository** (ADR) has been developed within context of AMBER Coordination Action project, where our research team is integrated and leading the work of development of the ADR [1]. The repository is aimed at collecting raw data results from resilience assessment experiments to promote interaction among research groups. The repository is an infrastructure that integrates data from different sources in such way that enables comparison and cross-exploitation in a meaningful manner.

The AMBER data repository does not target data information related to security assessing and testing tools. In fact, the ADR and our proposal combined endow research community with powerful mechanisms to increase the impact of research. The first collects raw data from resilience assessment experiments. The second collects information about the tools used in these experiments.

2.3. Software and Information Repositories

The importance of software and software information repositories has increased in the last years, given factors such as the users necessity of finding tools to test or to assess the security aspects of their systems. This subsection presents the most relevant of these initiatives we have found.

SourceForge (SF) is the world's largest repository of open source software. It provides "free hosting to projects with a centralized resource for managing project, issues, communications, and code" [20]. This is a powerful example of software repository that offers, among other things, mailing lists, message forums, and version controlling. One interesting aspect of SourceForge is the ability of performing advanced search in its database through a user-friendly web-interface. Although its significant contribution to the overall open source movement, SourceForge is not a repository for finding security assessment and testing tools. It is a generic repository for open-source development software, though containing some projects related to security assessment and testing tools. Examples of software related to security testing in SourceForge are Wapiti (to audit the security of your web applications) and W3AF (a Web Application Attack and Audit Framework).

The **Open Web Application Security Project** (OWASP) is "a worldwide free and open community focused on improving the security of application software" [21]. Besides the coordination of diverse initiatives such as the ranking of the most common web application security vulnerabilities (OWASP Top Ten Project) and the proposal of security testing procedures and checklists (OWASP Testing Guide), OWSAP also recommends the use of security assessment and testing tools, such as WebScarab and WZFuzzer. However, the list of recommendation does not present tools that are not taking part of the OWASP and does not offer means for comparing the security characteristics of the tools.

Sectools.org is perhaps the most important reference in terms of indexing network security tools [22]. This ranks the favorites tools of a security mailing list with thousands of users. The last ranking edition categorized the top tools in classes such as password crackers, sniffers, vulnerabilities scanners tools, web scanners, wireless scanners tools, vulnerabilities exploitation tools, and packet crafters. Each tool ranked in this web site is described by one or more of the following attributes: type of license (commercial or free), operating system (Linux, Windows, Mac OS X, etc.), type of user interaction (command-line or GUI), and source code availability. However, the descriptions of the tools are too generic and the users need to access the site of the vendor to find specific characteristics such as the type of target system that the tool address. One important limitation in this site is the lack of a comparison of the functionalities of the tools.

3. Repository architecture and underlying technology

SATTIR was developed using widely known open-source technologies. Under a model-view-controller architectural pattern (MVC), the repository uses the Java EE and Spring framework as the programming technologies. The selected web server was the Apache Tomcat 6. The data warehousing and OLAP techniques are currently implemented using

the Mondrian and JPivot frameworks (http://mondrian.pentaho.org/). These technologies are distributed in the tree main parts of SATTIR Architecture (Figure 1):

- The **Data Collection Infrastructure (#1)** is aimed at providing mechanisms and interface where repository users can submit information on tools. The infrastructure relies on the user interaction to collect new data through the completion of a proper form. For that reason, in the current architecture mechanisms are not necessary to extract transform and load information from any source. Examples of mechanisms in these infrastructures are the JSP pages coded to build the form where uses will complete information.
- The **Data Storage Infrastructure (#2)** is constituted by the database of the repository. The database management system supporting the repository is the PosgreSQL 8. The relation entity model of the database is described in details in the next section.
- The **Dissemination and Sharing Data Infrastructure (#3)** contains the mechanisms that present the repository contents to the end-user, using the repository web servers. One important aspect here are the mechanisms that answer the analysis questions users requests (data warehousing, statistical analysis, and information retrieval).

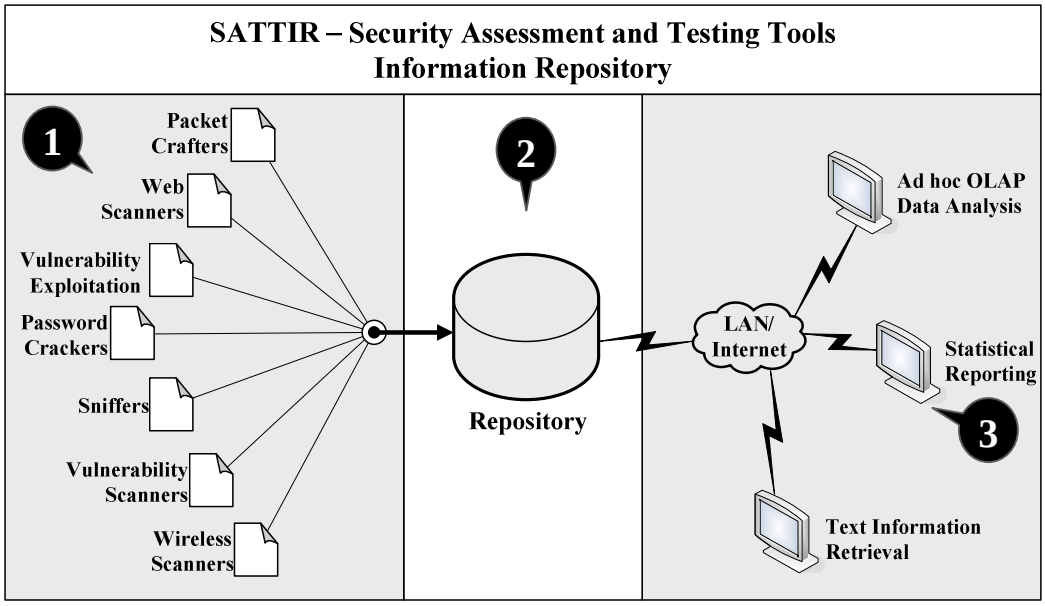


Figure 1. SATTIR Architecture

Business Intelligence (BI) and Information Retrieval technologies are part of the Dissemination and Sharing Data Infrastructure. In our repository, BI refers to multidimensional analysis, data warehousing and On-Line Analytical Processing.

A data warehouse is a global repository that stores large amounts of data that has been extracted and integrated from heterogeneous systems (operational or legacy systems) [23]. OLAP (On-Line Analytical Processing) is the technique of performing complex analysis over the information stored in a data warehouse [23]. The data

warehouse coupled with OLAP enables decision makers to creatively analyze and understand business trends since it transforms operational data into strategic decision making information.

In data warehousing the data is organized according to the multidimensional model (star model), which includes two kinds of data: facts and dimensions. Facts are numeric or factual data that represent a specific business or process activity and each dimension represents a different perspective for the analysis of the facts. Each dimension is described by a set of attributes. Note that the facts are just numerical quantities and only acquire meaning when are referenced to the dimensions. The OLAP analysis over a multidimensional cube consists of dicing and slicing the data in order to compute the desired measures. Figure 2 depicts one example of the data warehousing model within the context of our proposal, showing the fact table Citations in the center of the model. The entities around the fact table represent the model dimensions.

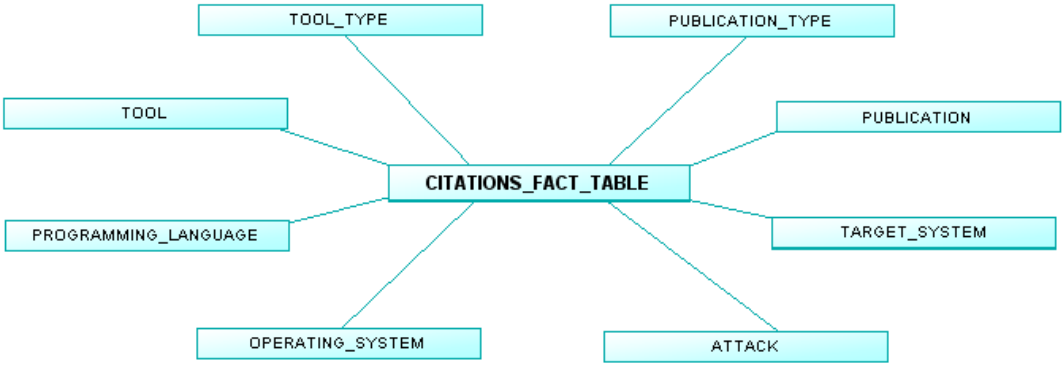


Figure 2. Example of data warehousing model

Information retrieval techniques will be used for raw data in textual formats that is not subject to a multidimensional characterization in facts and dimensions, nor even in a well defined set of attributes such as data used for data mining analysis. Textual data is indexed using traditional IR techniques based on inverted files and ranking algorithms to sort out search results by order of relevance. This way, the user can find specific patterns in the textual raw data, which is normally used to guide specific (and manual) data analysis of interesting cases (e.g., faults that lead to a unexpected behavior, not adequately handled by fault tolerance mechanisms available in the system).

4. Data storage infrastructure

The ability to provide meaningful comparisons of security tools depends essentially on the quality, the diversity, and the amount of information and the existence of a proper infrastructure to store and analyse that information.

Figure 3 presents a snapshot of SATTIR Class Diagram containing the most relevant entities of SATTIR. The main entity of this diagram, and of course of the repository, is the Tool Entity. This entity is the cornerstone of the repository, where all repository functionalities are based on, from storing to exploring information. The remainder entities represented in the class diagram are either related to intrinsic tool characteristics (represented outside the green box of Figure 3) or to extrinsic characteristics such as the academic publications related to a given tool. The intrinsic

tool characteristics are still differentiated by its relation with security aspects (represent by the red box) or not (all entities in the blue box except Tool Entity).

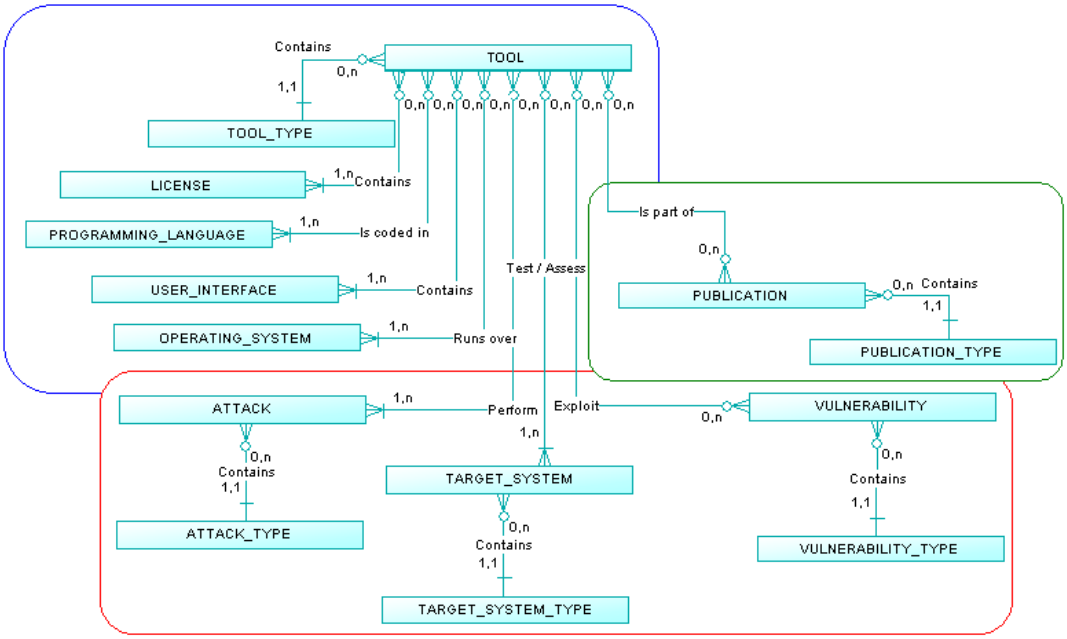


Figure 3. SATTIR Class Diagram

Another important aspect is to understand the type of information that can be stored in the repository entities. To this goal, we detail briefly the entities around the Tool Entity in Table 1 and Table 2. This main entity, therefore, contains attributes such as tool name, tool description, tool owner and other basics information of the tool.

Table 1. Description of SATTIR Entities (1/2)

Entity Scope	Entity Category	Entity Name	Entity Description
Internal	General	Tool Type	This contains values such as packet crafters, password crackers, sniffers, vulnerability exploitation, vulnerability scanner, web scanners, and wireless scanners.
		License	This refers either to commercial licenses or public licenses such as GNU General Public License (GPL), Academic Free License (AFL).
		Programming Language	This refers to the programming languages used in the coding of a given tool, containing values as Java, C++, and Python.
		User Interface	This refers to the means that user may interact with the tool, such as command-line and graphical interface.
		Operating System	This refers to the operating system that supports the installation of a given tool such as Windows, Linux, Unix, and Mac OS X.

With an in-depth focus on tool characterization and the proper design of the repository data storage infrastructure, it is possible to predict questions that may be queried to the repository. Two examples of meaningful questions that are answered by our model and that confirm the relevance of our approach are: How many citations a given type of tool received in a given period? Which are the tools that test or assess a given set of target system?

Table 2. Description of SATTIR Entities (2/2)

Entity Scope	Entity Category	Entity Name	Entity Description
Internal	Security	Target System	This refers to the system under test or evaluation such as web servers, web services, web applications, and wireless and network.
		Target Type	This contains values such as network and web systems.
		Attack Type	This refers to the category of attacks addressed by a given tool, such as Injection, Protocol Manipulation, and Sniffing Attacks [10].
		Attack	This contains values such as Cross-site scripting, SQL Injection, Buffer Overflow.
		Vulnerability Type	This refers to the category of vulnerabilities that a given tool exploits using its attacking functionalities. This contains values such as Access Control Vulnerability, Protocol Errors, and Environmental Vulnerability [10].
		Vulnerability	This contains values such as Insufficient Privileges, User Management Errors, and Empty String Password.
External	Citations	Publication Type	This contains values such as Magazine, Journal, and Proceeding Conferences.
		Publication	This refers to the description of the magazine, journal or conferences where a given tool was cited.

5. Potential users and exploitation scenarios

The main motivation of SATTIR is to be simple and fast, contributing to reduce radically the effort of users in finding, analyzing, and comparing security assessment and testing tools. To this goal, the repository can be used in two typical scenarios:

- As a form of common repository to store and share tool information. The current situation is not satisfactory as tool information is spread by different sources. SATTIR can change drastically this situation, as it uses a common format to share the data with the advantage of using multidimensional model. At the same time, the availability of tool information in SATTIR allows a very efficient dissemination of the tools recently proposed by industry and academic communities. **End-users**, **practitioners** and **researchers** may found in the repository the state-of-the-art of tools to test and evaluate the security of their systems and to employ in their experimental setups.
- To perform fast analysis of tool information in a very efficient way, ranging from cross-exploitation to comparative perspective. **Tool vendors** can analyze the strengths and the weakness of competing tools in order to improve the features of their own tools. **Researchers** will identify the research impact of a given tool through the analysis of the number of citations that it received in a given academic context. Vendors and researchers may still identify the necessity of proposing new tools to cover aspect still not addressed by the existents tools.

In practice there are two types of potential users: readers and writers. Readers access the repository only to explore and analyze tool information. Writers contribute to the repository by making tool information available to the community. Obviously, writers also use the repository to analyze their information, compare it to information provided by other repository contributors, and disseminate the data worldwide.

Henceforth, repository readers will be referred as **SATTIR Readers** and writers as **SATTIR Contributors**.

The simplicity of SATTIR architecture is emphasized through the tree steps needed to use the repository:

1. **User registration and authentication.** Before being able to access the repository, either SATTIR Readers or SATTIR Contributors have to register in the repository web site. Registration is then analysed and approved. Authentication is required every time users access the data repository. The registration is conditioned to the acceptance of the terms and conditions of the repository. The registration is required in order to allow future statistical analysis (e.g. the most compared set of tools in a given category of potential users).
2. **Submission of tool information.** After registration, SATTIR Contributors are able to send their information through the web interface of SATTIR. The availability of the submitted information in the repository relies on the approval of SATTIR Team. While SATTIR Team evaluates the authenticity of the provided tool information, which may take from 1 to 3 business days, the information will be set as pending state. The access to the pending states tools is just available to SATTIR Contributors. This aims to avoid duplicate efforts in order to register a given tool. However, contributors may suggest the improvement of the description of a given tool.
3. **Analysis of the information** using the tools and techniques (OLAP and Information Retrieval) available in the repository. This includes not only the normal cross-exploitation of tool information but also comparative analysis using multidimensional approaches to find out the differences among tool that addresses a given set of target system.

6. Functionalities overview

The preliminary version of SATTIR web site is currently available at http://security.dei.uc.pt/sattir. This web site was developed to provide easy access to the community to the repository functionalities.

The potential users of the repository must become registered users in order to get access to the main functionalities of the repository such as the exploration of third-party data (readers and contributors) and the submission of data (contributors). The relationship of registered users to the repository may be of several types: end-users, practitioners, makers, and researchers.

The interface of SATTIR web site (Figure 4) is constituted of the following sections:

Menu section (#1). The menu section allows the access to the main functionalities of the repository (e.g. tool exploration, tool submission, user registration, documentation).

Direct access section (#2). This section presents the authentication fields that users must fill to obtain access to protected repository functionalities (e.g. tool submission).

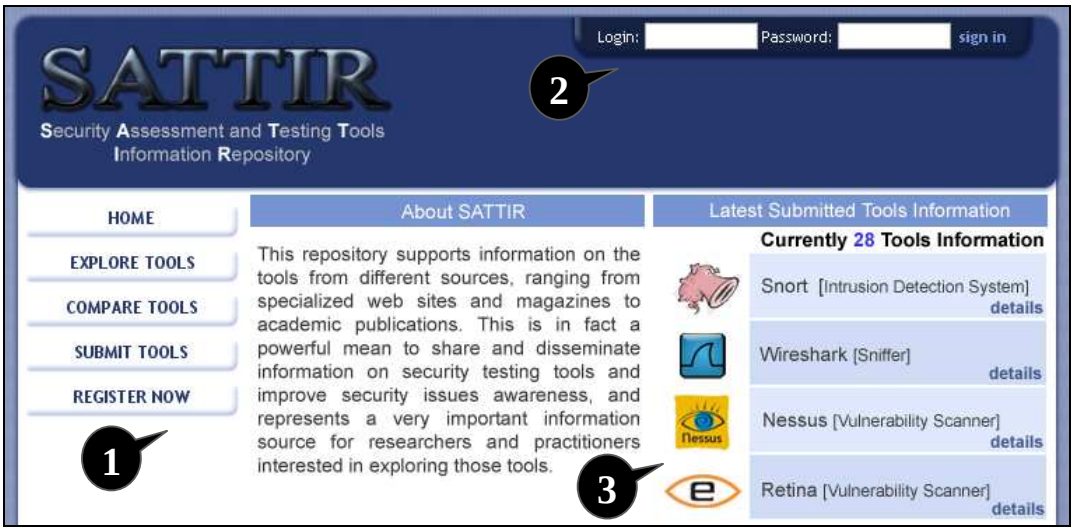


Figure 4. Home Page of SATTIR

Announce section (#3). This section shows the most recent updates in the repository such as the latest submitted tools information.

The main functionalities of repository web site are available through the menu section. The high importance of these functionalities leads us to describe the most relevant of them:

- **Submit tool.** This option allows the user to contribute data about a particular tool. The precondition here is to have an account with privileges of contributor. After the registration, the new repository member is able to send the characteristic of the tools to the repository through the completion of a proper form.
- **Explore tool**. This option provides the access to the most meaningful functionalities of the repository. This option offers to users the list of all tools that are available for analysis, showing a detailed description of those tools. Users also may select several tools and click on the compare button to see the characteristics of the tools side-by-side. Furthermore, if a given class of tools is meaningful for the user, there is an option where the user may check to receive information by email when new information is available.

To help disseminating the data, an up-to-date log of data addition and modification is posted in the web site focusing on the most recent additions or modifications. A mechanism for email notifications is also available. For example, users can request a notification to be sent when a given dataset is updated (data added or modified) or when a new dataset related to a particular research field is created.

7. Conclusion and Ongoing Work

In this paper we presented the **Security Assessment and Testing Tools Information Repository** (SATTIR). This repository provides to the community an infrastructure to collect and explore information about network security assessment and testing tools. It also offers useful functionalities to the community (end-users, practitioners, researchers,

and makers) through the indexation of tools information from different sources supporting the contribution of users, the comparison of tools characteristics using a relational database approach, the support of advanced information analysis using information retrieval and data warehousing, and the dissemination and sharing of tools information using web-enabled tools.

In fact, SATTIR is a potential tool for industry and research community to disseminate their security testing tools and prototypes. An important aspect is that, if successful, SATTIR will contribute to increase tools sales (in the case of commercial tools) and to augment the number of citations to the research papers describing the prototypes (in the case of research prototypes). To contribute to the repository, users just need to register and submit information about tools. The analysis of existing data is open to the entire world. The repository is maintained by an experienced team that will help users using the repository.

Ongoing work includes extending the current functionalities of the repository as well as adding data on more tools. The representativeness of this repository will also rely on the ability of storing and comparing the most meaningful tools. For that reason, we expect vendors, security and research community to provide data from network security assessment tools to the repository.

References

[1] AMBER Data Repository, available from http://amber.dei.uc.pt/repository/; Internet; accessed 29 March 2008.

[2] Crow, R. "A Guide to Institutional Repository Software.", Open Society Institute, 2004.

[3] Wyles, R., et al., "Technical Evaluation of selected Open Source Repository Solutions", Open Access Repositories in New Zealand (OARINZ) project, Christchurch Polytechnic Institute of Technology, v1.3, 13 September 2006.

[4] Eprints for Digital Repositories, available from http://www.eprints.org/; Internet; accessed 29 March 2008.

[5] DSpace, available from http://www.dspace.org; Internet; accessed 30 March 2008.

[6] Lucene Search Engine, available from http://lucene.apache.org/; Internet; accessed 30 March 2008.

[7] Payette, S., Lagoze, C., "Flexible and Extensible Digital Object and Repository Architecture (FEDORA)," Second European Conference on Research and Advanced Technology for Digital Libraries, Heraklion, Crete, Greece, September, 1998.

[8] Fedora Commons, available from http://www.fedora-commons.org; Internet; accessed 29 March 2008.

[9] Repository in Box, available from http://icl.cs.utk.edu/rib/; Internet; accessed 30 March 2008.

[10] Open Repository, available from http://www.openrepository.com/; Internet; accessed 30 March 2008.

[11] Proquest Digital Commons, available from http://www.proquest.com/; Internet; accessed 30 March 2008.

[12] DACS, Software Reliability Dataset, available from https://www.thedacs.com/databases/sled/swrel.php; Internet; accessed 15 March 2008.

[13] Delude, J., Vienneau, R., “Analyzing Quantitative Data Through the Web”, Proc. Of the 6th Annual Dual Use Technologies & Applications Conference, June 1995.

[14] Nelson, R., “Software Data Collection and Analysis”, Rome Air Development Center, Rome, NY, September 1978.

[15] Musa, J., “Software Reliability Data”, Data & Analysis Center for Software, January, 1980.

[16] NASA/WVU IV&V Facility, Metrics Data Program, available from http://mdp.ivv.nasa.gov; Internet; accessed 15 March 2008.

[17] Error, Fault, and Failure Data Collection and Analysis, available from http://hissa.nist.gov/project/eff.html; Internet; accessed 15 March 2008.

[18] Schroeder, B., Gibson, G., “The Computer Failure Data Repository (CFDR)”, Workshop on Reliability Analysis of System Failure Data (RAF'07), MSR Cambridge, UK, March 2007.

[19] The computer failure data repository (CFDR), available from http://cfdr.usenix.org/; Internet, accessed 25 March 2008.

[20] SourceForge, available from http://sourceforge.net/projects/repository/; Internet; accessed 29 March 2008.

[21] Open Web Application Security Project, available from http://www.owasp.org/index.php/Category:OWASP_Project; Internet; accessed 24 March 2008.

[22] Top 100 Network Security Tools, available from http://sectools.org/; Internet; accessed 24 March 2008.

[23] Chauduri, S., Dayal, U., “An overview of data warehousing and OLAP technology”, SIGMOD Record, March 1997.

Detectores Perfeitos em Sistemas Distribuídos Não Síncronos

Raimundo José de Araújo Macêdo[1], **Sérgio Gorender**[1]

[1]Laboratório de Sistemas Distribuídos (LaSiD)
Departamento de Ciência da Computacao
Universidade Federal da Bahia
Campus de Ondina - Salvador - BA - Brasil

{macedo,gorender}@ufba.br

Abstract. *In this paper we show that is possible to implement a perfect failure detector P (one that suspect all faulty processes if and only if those processes failed) in a system weaker than the synchronous system (contradicting a common believe). To realize that, we introduce the partitioned synchronous system (Spa) that is strictly weaker than the conventional synchronous system (it is not always possible to implement global synchronous computations in Spa such as internal clock synchronization). From some properties we introduce (such as strong partitioned synchrony), we show how to implement P over Spa. Moreover, we show that even if strong partitioned synchrony cannot be sustained, we still are able to take advantage of the existing synchronous partitions to improve the robustness of applications, by introducing a partially perfect detector named xP. The necessary properties and algorithms to implement P and xP are presented in the paper, as well as the related correctness proofs.*

Resumo. *No presente artigo mostramos que um detector perfeito de defeitos P(que suspeita todos os processos que falharam se e somente os mesmos falharam) pode ser implementado num sistema mais fraco que o sistema distribuído síncrono (contrariando uma crença estabelecida). Nesse sentido, introduzimos o sistema síncrono particionado (Spa) que é estritamente mais fraco que o sistema síncrono (em Spa não é sempre possível implementar ações síncronas globais como sincronização interna de relógios). Através da propriedade que definimos como sincronia particionada forte, mostramos como implementar P em Spa. Melhor ainda, mostramos que mesmo que Sincronia Particionada Forte não possa ser garantida, podemos ainda assim tirar proveito das partições síncronas existentes para melhorar a robustez das aplicações de tolerância da falhas, através de um detector parcialmente perfeito, denominado por nós de xP. As propriedades e algoritmos necessários para implementar P e xP são introduzidos no artigo, assim como as provas de correção relacionadas.*

1. Introdução

A capacidade de resolver certos problemas de tolerância a falhas em sistemas distribuídos está intimamente ligada à existência de um modelo de sistema adequado. Nesse sentido, há algumas décadas, pesquisadores vêm propondo uma variedade de modelos de resolução de problemas, onde os modelos assíncronos (ou livres de tempo) e síncronos (baseados no tempo) têm dominado o centro das atenções, por serem considerados modelos extremos em termos de

resolução de problemas de tolerância a falhas. Por exemplo, o problema de difusão confiável - na presença de canais confiáveis e falhas silenciosas de processos - é solúvel em ambos os modelos [Lynch 1996, Mullender 1993]. Contudo, o problema de consenso distribuído é solúvel no modelo síncrono, mas não no modelo assíncrono [Fisher et al. 1985]. A origem da não solubilidade do consenso em modelos de sistemas assíncronos está relacionada à dificuldade em distinguir-se, nesse modelo, entre uma falha de um processo p (exemplo, de parada) e o atraso no recebimento de uma mensagem de p. Essa crença, levou pesquisadores a acreditarem que o poder de um modelo de sistema assíncrono equipado com um detector perfeito de falhas - que identifica todos os processos que falharam, se e somente se os mesmos falharam - se igualaria ao poder de um modelo de sistema síncrono.

Essa crença foi adequadamente contestada por Charron-Bost, Guerraoui e Schiper [Charron-Bost et al. 2000], que mostraram que o modelo de sistema síncrono (denominado por eles Ss) é estritamente mais forte que o modelo de sistema assíncrono equipado com um detector perfeito (denominado por eles como Sp). No referido artigo, eles demonstraram que alguns problemas solúveis em Ss não o são em Sp, mesmo quando tais problemas não envolvam especificações temporais (nesses casos, por definição, impossíveis de serem resolvidos nos modelos assíncronos). Alem disso, eles provaram que a solução do problema de consenso distribuído uniforme - um problema fundamental de tolerância a falhas - é feita com mais eficiência em Ss, quando comparado a Sp, usando como métrica o grau de latência, que é baseada no conceito de número de rodadas [Schiper 1997].

Por outro lado, ao longo dos anos, a implementação de detectores perfeitos, que tornaria o problema de consenso solúvel, tem sido considerada impossível em sistemas assíncronos. Esse fato levou a Chandra e Toueg [Chandra and Toueg 1996] a proporem um modelo de computabilidade para tolerância a falhas em sistemas assíncronos baseado em propriedades axiomáticas de várias classes de detectores de defeitos, identificando aquelas para as quais existem soluções com os menores requisitos de implementação (no caso, a classe $\diamond W$). A partir daí, vários autores passaram a propor soluções de implementação para essas classes [Chen et al. 2000, Macêdo 2000, Nunes and Jansch-Pôrto 2004, Falai and Bondavalli 2005, Lima and Macêdo 2005] cuja viabilidade de funcionamento dependeria de certas condições de estabilidade do ambiente de execução assíncrono (exemplo, a propriedade Global-Stabilization-Time [Dwork et al. 1988]). A classe mais forte apresentada por Chandra e Toueg foi justamente a do detector perfeito, denominado por eles de P e possuindo as propriedades axiomáticas de *strong completeness* (P suspeita todos os processos que falharam) e *strong accuracy* (P somente suspeitam processos que realmente falharam). No mesmo artigo, Chandra e Toueg argumentam que seria possível implementar P no modelo síncrono de sistemas distribuídos, utilizando-se de *timeouts*.

Baseados na idéia de que *timeouts* ou hipóteses temporais podem ser utilizados nos sistemas Ss para implementar detectores de falhas perfeitos, Charron-Bost, Guerraoui and Schiper prosseguem em seu artigo [Charron-Bost et al. 2000] para concluir que a comparação dos dois modelos se daria via a identificação das propriedades do modelo Ss que se perderiam na transformação axiomática proposta por Chandra e Toueg. Apesar de elucidativas, tais discussões relativas ao artigo de Charron-Bost, Guerraoui e Schiper incorrem em uma imprecisão: a crença subjacente de que o ambiente de implementação de ambos os modelos, Ss e Sp, seria necessariamente o sistema síncrono[1] onde existem limites temporais

[1]Neste artigo, *sistema* refere-se ao ambiente de execução e *modelo* às propriedades que se pode inferir de

superiores conhecidos para processamento de informação e entrega de mensagens nos canais de comunicação (observem que tal comportamento casa perfeitamente com o que se assume para modelo síncrono de sistema distribuído). Tal crença subjacente de fato justificaria a comparação vantajosa do modelo Ss em relação ao modelo Sp, como colocada no artigo. Tal imprecisão, de outro lado, reforça outra crença: a de que detectores perfeitos não seriam implementáveis em sistemas mais fracos que os síncronos.

No presente artigo nós desmontamos essa última crença mostrando que P pode ser implementado num sistema mais fraco que Ss, denominado por nós de Spa (síncrono particionado), o que torna injusta a comparação direta do poder de resolução dos dois modelos Ss e Sp, uma vez que Ss requer mais recursos para sua implementação comparada ao modelo Sp (quando implementado sobre Spa). Vale salientar, como veremos adiante, que o modelo Sp implementado sobre Spa não requer a existência de um *wormhole* síncrono [Veríssimo and Casimiro 2002] ou *spanning tree* síncrona [Gorender and Macêdo 2002], onde seria possível implementar ações síncronas globais em todos os processos, como sincronização interna de relógios. No sistema Spa, que propomos, componentes do ambiente distribuído necessitam ser síncronos, mas os mesmos não precisam estar conectados entre si via canais síncronos, o que torna impossível a execução de ações síncronas distribuídas em todos os processos do sistema. Mesmo assim, mostramos ser possível implementar P em Spa. Melhor ainda, mostramos que mesmo que parte dos processos não esteja em qualquer das componentes síncronas, podemos ainda assim tirar proveito das partições síncronas existentes para melhorar a robustez das aplicações de tolerância a falhas, através de um detector parcialmente perfeito, denominado por nós de xP.

O resto deste artigo se estrutura da seguinte forma. Na Seção a seguir são apresentados trabalhos relacionados. Na Seção 3 é apresentado o modelo Spa, suas características e propriedades. As Subseções 3.1, 3.2 e 3.3 descrevem uma implementação de um detector de defeitos P no sistema Spa, e apresentam suas propriedades e provas formais, e na Subseção 3.4 é apresentada a classe de detectores de defeitos xP, com suas propriedades, além de uma implementação e provas formais. A Seção 4 apresenta as conclusões ao trabalho.

2. Trabalhos Relacionados: Modelos de Sistemas Distribuídos

O desenvolvimento de serviços tolerantes a falhas, como consenso distribuído, depende sobremaneira da existência de tempos conhecidos máximos para escalonamento de processos e transmissão de mensagens (isto é, canais e processos isócronos). Tais limites superiores temporais - para todos os processos e canais de comunicação - somente podem ser continuamente garantidos nos sistemas síncronos. Portanto, problemas como o consenso distribuído são solúveis em sistemas distribuídos síncronos [Lamport et al. 1982]. De outro lado, serviços de consenso distribuído podem também ser garantidos em ambientes assíncronos ou parcialmente síncronos desde que o "comportamento síncrono"se estabeleça durante períodos de tempo suficientemente longos para a execução do consenso [Dwork et al. 1988]. Assim sendo, tais ambientes considerados parcialmente síncronos podem apresentar comportamento alternado, entre síncrono com garantias temporais e assíncrono com nenhuma garantia temporal. Conseqüentemente, podemos considerar tais sistemas como híbridos na dimensão temporal. Por exemplo, o sistema assíncrono temporizado depende de períodos de estabilidade síncrona suficientemente longos para prover serviços, e o sistema pode alternar

determinado *sistema*

entre estável e não estável (i.e., síncrono e assíncrono) [Cristian and Fetzer 1999]. A hipótese GST - Global Stabilization Time - é necessária para garantir que os protocolos de consenso baseados no detector de defeitos $\diamond\mathcal{S}$ trabalhem de forma adequada [Chandra and Toueg 1996] (i.e., para trabalhar de forma correta o sistema deve apresentar comportamento "síncrono"a partir de algum momento).

Há outros modelos que consideram a natureza híbrida do ambiente não somente na dimensão temporal (como os anteriormente referenciados), mas também na dimensão espacial, onde o sistema é particionado em componentes síncronas e assíncronas que funcionam de forma concomitante. Logo, nós consideramos esses modelos como híbridos na dimensão espacial. Esse é o caso do modelo TCB que se utiliza de um wormhole síncrono (i.e., uma componente síncrona que inclui todos os processos do sistema) para implementar serviços tolerantes a falhas [Veríssimo and Casimiro 2002], e tal caráter híbrido persiste durante toda a vida do sistema.

No caso dos sistemas síncronos particionados (Spa) assumimos que o sistema computacional distribuído é capaz de prover qualidades de serviços fim-a-fim distintas tanto para execução dos processos quanto para os canais de comunicação. Nos sistemas operacionais, tais características podem ser implementadas através de sistemas operacionais de tempo real capazes de lidar ao mesmo tempo com tarefas críticas (com deadlines garantidos) e não críticas (*best-effort*). Nas redes de computadores, o mesmo efeito pode ser conseguido quando determinado nó está ligado a duas redes de QoS distintas (por exemplo, de um lado uma rede de tempo real e do outro uma rede TCP/IP). Também se pode conseguir tal característica através de multiplexação, de modo que um determinado canal físico possa ser capaz de transmitir fluxos de comunicação relacionados a diversos canais virtuais com qualidades de serviço (QoS) distintas (por exemplo, um canal isócrono e dois não isócronos implementados no mesmo canal físico). Tal efeito pode ser implementado através de redes de tempo reais híbridas ou arquiteturas de qualidade de serviços (e.g., diffserv [Aurrecoechea et al. 1998]).

Dado que canais e processos podem ser isócronos e não isócronos no mesmo sistema e ao mesmo tempo, consideramos nosso sistema híbrido no espaço. Não obstante, os canais não isócronos podem apresentar comportamento síncrono em certos períodos de estabilidade do sistema. Logo, nosso sistema Spa se diferencia dos demais modelos apresentados acima não somente porque considera o caráter híbrido nas dimensões temporal e espacial (nesse sentido, similar ao TCB [Veríssimo and Casimiro 2002]), mas também porque tal caráter híbrido pode mudar com o tempo, com períodos com poucas ou mesmo nenhuma partição síncrona (nesse caso, distinto do TCB). Em outras palavras, o sistema pode se tornar totalmente assíncrono ou não ter todos os seus processos em componentes síncronas, dado que em algumas circunstâncias a qualidade de serviço de canais ou processos pode degradar (devido a falhas, por exemplo).

Resumidamente, modelos de sincronia parcial como o GST e o assíncrono temporizado não consideram o caráter híbrido espacial. O modelo TCB é híbrido em espaço e tempo, mas sua natureza híbrida permanece durante a vida do sistema. Nosso modelo Spa considera um ambiente computacional subjacente que pode também ser híbrido no tempo e espaço. Contudo, a parte síncrona pode se deteriorar com o tempo. O preço que se paga para dispor de tal flexibilidade no modelo é a necessidade de implementação de mecanismos de gerenciamento de qualidade de serviço (provisão e monitoria) ambos nos sistemas operacionais e nas redes, capazes de acompanhar continuamente o estado de QoS dos processos e canais. Tal mecanismo é, em certo sentido, semelhante ao serviço *fail-awareness*

implementado no sistema assíncrono temporizado, onde a ocorrência de certo número de falhas de desempenho ativa um sinal indicando que o sistema não mais pode satisfazer ao comportamento "síncrono"[Fetzer and Cristian 1996]. A principal diferença reside no fato de que no sistema assíncrono temporizado a sinalização não é baseada na gestão de QoS e sim na violação de limites de tempo para envio de recebimento de mensagens (*round-trip times*).

3. O Modelo Síncrono Particionado Spa

Modelo de sistema e hipóteses assumidas Um sistema é formado por um conjunto Π de processos $p_1, p_2, ..., p_n$, e um conjunto χ de canais de comunicação $c_1, c_2, ..., c_m$. Cada canal c_i de χ liga somente dois processos distintos de Π e cada processo de Π está ligado a todos os outros processos de Π. Portanto, nosso sistema forma um grafo simples completo não direcionado $DS(\Pi, \chi)$ com $(n \times (n-1))/2$ arestas. Tanto processos em Π quanto canais em χ podem ser isócronos ou não isócronos. Um dado processo é isócrono se existe valor máximo conhecido (digamos, ϕ) para a execução de passos de computação em p_i. Da mesma forma, um canal c_i é isócrono se uma mensagem é transmitida em c_i dentro de um limite de tempo limitado e conhecido, digamos δ. Do contrário, processos e canais são não isócronos. δ e ϕ são parâmetros do sistema de execução e garantidos por mecanismos operacionais de sistemas e redes de tempo real. Assumimos também que processos e canais podem mudar sua qualidade de serviço de forma que possam alternar entre isócronos e não isócronos.

Assumimos que canais são confiáveis, ou seja, não alteram ou perdem mensagens enviadas. Também assumimos que processos falham de forma silenciosa, ou seja, apenas param prematuramente de funcionar, sem produzir quaisquer efeitos adicionais. Processos considerados corretos, não falham durante um intervalo de tempo de interesse (por exemplo, durante a execução da aplicação).

Dados $\Pi' \subseteq \Pi$, $\Pi' \neq \emptyset$ e $\chi' \subseteq \chi$, $\chi' \neq \emptyset$, um componente ou sub-grafo $C(\Pi', \chi') \subseteq DS(\Pi, \chi)$ conexo é síncrono se $\forall p_i \in \Pi'$ e $\forall c_j \in \chi'$, p_i e c_j são isócronos. Utilizamos a notação Cs para denotar um componente C síncrona e Ca para um componente não síncrono.

A Figura 1 ilustra a representação no grafo DS para duas redes locais com propriedades síncronas (processos e canais de comunicação), interligadas por um canal assíncrono (por exemplo, um canal TCP/IP). No exemplo, cada nó da rede (númerados de 1 a 6) hospeda apenas um processo. Observa-se em DS, por exemplo, que os sub-grafos $C1(\{1,2,3\}, \{(1,2),(1,3),(2,3)\})$, $C2(\{4,5\}, \{(4,5)\})$ e $C3(\{5,6\}, \{(5,6)\})$ formam componentes síncronos, enquanto que $C4(\{3,4,5\}, \{(3,4),(4,5),(3,5)\})$ forma um componente não síncrono.

No sistema distribuído Spa, assumimos inicialmente a seguinte propriedade necessária para a implementação de P.

- *Sincronia Particionada Forte*: $\forall p_i \in \Pi$, $\exists$ um $Cs \subset DS$ tal que $p_i \in Cs$.

Na exemplo da Figura 1, *Sincronia Particionada Forte* se verifica pois $C1$, $C2$, e $C3$ incluem todos os processos em DS. A hipótese de *Sincronia Particionada Forte* será relaxada mais tarde no texto quando introduzirmos o detector xP, onde processos podem pertencer a componentes não síncronos (Ca). No entanto, é importante observar que para um dado sistema distribuído, *Sincronia Particionada Forte* não necessariamente implica que exista um componente Cs onde $\Pi = \Pi'$, o que caracterizaria um *wormhole* síncrono na terminologia

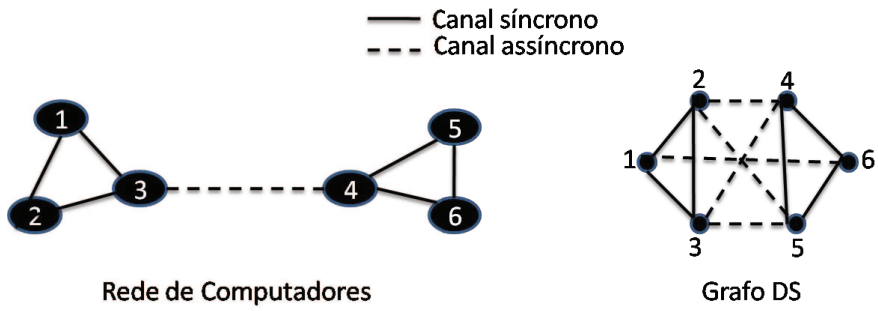


Figura 1. Exemplo de uma Rede e sua representação no Grafo DS

em [Veríssimo and Casimiro 2002] ou *spanning tree* síncrona (no exemplo da Figura 1 não há qualquer sub-grafo Cs que inclua todos os vértices de DS).

Assumimos, também, que seja possível, a partir do ambiente de execução, identificar localmente qual a qualidade de serviço de um dado processo ou canal de comunicação.

3.1. Detector Perfeito em Spa

Descrevemos a seguir a implementação de um detector de defeitos que satisfaz as propriedades *strong completeness* e *strong accuracy*, para detectores de defeitos da classe P, quando executando em Spa com a propriedade *Sincronia Particionada Forte*. Esta implementação foi inicialmente apresentada em [Macêdo et al. 2005, Gorender et al. 2007].

Informações com relação ao estado dos processos, e à possibilidade de se efetuar detecções confiáveis, é fornecida ao detector de defeitos por dois mecanismos adicionais, chamados Detector de Estados dos Processos e QoS Provider [Gorender 2005, Macêdo et al. 2005]. A partir das informações providas por estes mecanismos o detector de defeitos verifica as condições mínimas para assumir um comportamento de um detector Perfeito.

O QoS Provider (QoSP) funciona como uma interface entre o ambiente de execução e os Detectores de Defeitos e de Estados, e basicamente informa a estes detectores quais canais de comunicação são isócronos e quais não são. Esta informação é fornecida através da execução da função $QoS(p_i, p_j)$, onde p_i, p_j identifica o canal de comunicação ligando os processos p_i e p_j. A função identifica o canal como T, quando o canal é isócrono, e U, quando o canal não é isócrono. Uma descrição detalhada do QoSP pode ser encontrada em [Macêdo et al. 2005].

O Detector de Estados dos Processos executa distribuído em módulos, um para cada processo do sistema, e detecta o estado de cada processo como sendo *live* ou *uncertain*, inserindo a identificação destes processos nos conjuntos $live_i$ e $uncertain_i$, fornecidos a cada processo p_i do sistema. Processos que fazem parte de uma Componente Síncrona (Cs), são processos isócronos que possuem ao menos um canal de comunicação isócrono, e são identificados no estado *live*. Este mecanismo utiliza informações do QoS Provider, sobre os canais de comunicação. O Detector de Estados satisfaz as propriedades *strong completeness* (toda transição no estado de um processo entre $live$ e $uncertain$ será detectada por todos os processos corretos do sistema, em um tempo finito) e *strong accuracy* (nenhuma detecção de uma transição de estado entre $live$ e $uncertain$ para um processo, efetuada por qualquer processo correto do sistema, será incorreta.). A descrição detalhada deste mecanismo, além

das provas das propriedades, pode ser encontrada em [Gorender 2005, Macêdo et al. 2005, Gorender et al. 2007].

Como em Spa com a propriedade **Sincronia Particionada Forte** todos os processos pertencem a alguma Componente Síncrona, temos que o Detector de Estados dos Processos identifica todos os processos como possuindo o estado *live*: $\forall p_i \in \Pi, live_i = \Pi$.

3.2. Algoritmo do Detector de Defeitos

O detector de defeitos é executado em módulos, um para cada processo do sistema. Os módulos do detector de defeitos enviam periodicamente mensagens *are-you-alive* para todos os demais módulos do detector de defeitos, solicitando destes, em resposta, o envio de mensagens do tipo *I-am-alive*, através dos canais de comunicação, segundo o modelo *pull*. O tempo esperado para as respostas dos processos (*timeout*), em cada canal de comunicação $c_{x/y}$, é definido pela função $Delay(p_x, p_y)$ do QoS Provider. No caso de canais de comunicação isócronos a função $Delay(p_x, p_y)$ informa limites de tempo conhecidos para a transferência de mensagens. Portanto, para estes canais, o módulo do detector de defeitos provê notificações de falhas de processos que de fato tenham falhado. Para canais não isócronos, os módulos do detector de defeitos produzem suspeitas de falhas, ou seja, detecções não confiáveis de falhas de processos. Detecções confiáveis são registradas com a inserção da identificação do processo detectado no conjunto $down$, e sua retirada do conjunto $live$.

O algoritmo de detecção, apresentado na Figura 2, monitora os processos do sistema. Este procedimento é composto de 5 tarefas.

A primeira tarefa do algoritmo de detecção ($task1$) efetua o envio de mensagens $are - you - alive(p_i)$ para todos os processos do sistema (linha 3). Estas mensagens são enviadas periodicamente, a cada intervalo de monitoramento, indicado pela variável $MonitoringInterval$. Ao enviar cada mensagem *are-you-alive*(p_i), a tarefa calcula um *timeout* para o recebimento da mensagem *I-am-alive*(p_j) em resposta, para cada processo p_j do sistema. Os valores de *timeout* são armazenados no *array* $timeout_i$, indexado pela identificação do processo monitorado p_j. Estes tempos são calculados somando o tempo atual, obtido do relógio local, com o RTT calculado para o canal $c_{i/j}$, obtido através da função $Delay(p_i, p_j)$, do QoS Provider, com um valor de erro ($timeout_i[p_j] = CT_i() + Delay(p_i, p_j) + \alpha$ na linha 2).

A tarefa seguinte deste algoritmo ($task2$) efetua o monitoramento dos processos do sistema. Sempre que o $timeout$ para um processo não notificado (processo que não pertença ao conjunto $down_i$) seja alcançado (tempo obtido do relógio local for superior ao do *timeout*) sem o recebimento de uma mensagem $I-am-alive(p_j)$ do processo monitorado, este processo será analisado. Se o processo monitorado pertencer ao conjunto $live_i$ e o canal de comunicação utilizado para a monitoração for isócrono (linhas 5 e 6), o processo monitorado (p_j) tem sua falha notificada (linhas 7 a 9). Para verificar se o canal $c_{i/j}$ é de fato isócrono é utilizado o grafo DS e a função $QoS(p_i, p_j)$ do QoS Provider. A Função $QoS(p_i, p_j)$ é utilizada para se ter certeza de que um canal de comunicação identificado no grafo DS como isócrono permanece isócrono, uma vez que os canais podem ter a sua *QoS* alterada durante a execução. Se o canal de comunicação utilizado para a monitoração não for isócrono, nada é realizado (linha 10). Se o processo monitorado pertencer ao conjunto $uncertain_i$ (linha 12), terá sua identificação inserida no conjunto $suspected_i$ (linha 13).

```
     Task T1: every monitoringInterval do
(1)          for_each p_j, p_j ≠ p_i do
(2)                   timeout_i[p_j] ← CT_i() + Delay(p_i, p_j) + α;
(3)                   send "are-you-alive?" message to p_j
(4)          end_do

     Task T2: when ∃p_j : (p_j ∉ down_i) ∧ (CT_i() > timeout_i[p_j])) do
(5)          if ((p_j ∈ live_i)
(6)            then if ((DS_i[p_i, p_j] = timely) ∧ (QoS(p_i, p_j) = timely))
(7)                   then down_i ← down_i ∪ {p_j};
(8)                        live_i ← live_i − {p_j};
(9)                        send notification (p_i, p_j) to every p_x such that p_x ≠ p_i ∧ p_x ≠ p_j
(10)                  else do nothing (wait for a remote notification)
(11)                 end_if
(12)           else if ((p_j ∈ uncertain_i) ∧ (p_j ≠ suspected_i))
(13)                  then suspected_i ← suspected_i ∪ {p_j} end_if
(14)         end_if

     Task T3: when "I-am-alive" is received from p_j do
(15)         timeout_i[p_j] ← ∞; /* cancels timeout */
(16)         if (p_j ∈ suspected_i) then suspected_i ← suspected_i − {p_j} end_if
     Task T4: when notification(p_x, p_j) is received do
(17)         if p_j ∉ down_i then down_i ← down_i ∪ {p_j};
(18)                              live_i ← live_i − {p_j}
(19)         end_if

     Task T5: when "are-you-alive?" is received from p_j do
(20)         send "I-am-alive" to p_j
```

Figura 2. Algoritmo do módulo do detector de defeitos de (p_i)

A terceira tarefa (*task 3*) é executada sempre que uma mensagem $I - am - alive(p_j)$ é recebida pelo detector de defeitos. Se o processo que enviou a mensagem for um processo suspeito ($p_j \in suspected_i$), ele será retirado do conjunto $suspected_i$ (linha 16). Esta situação caracteriza uma suspeita errônea de uma falha.

A tarefa 4 (*task* 4) é executada quando uma mensagem $notification(p_x, p_j)$ for recebida pelo detector de defeitos, indicando a falha de um processo (p_x notifica a falha de p_j). Ao receber esta mensagem, o módulo do detector de defeitos insere a identificação do processo notificado (p_j) no conjunto $down_i$, caso ainda não seja um elemento deste conjunto (linha 17). A identificação do processo notificado é retirada do conjunto $live_i$ (linha 18).

A última tarefa (*task* 5) é executada quando uma mensagem *are-you-alive*(x, p_j) enviada pelo processo p_j é recebida pelo detector de defeitos. É enviada em resposta uma mensagem *I-am-alive*(x, p_i) para o processo p_j.

3.3. Propriedades e provas para o detector perfeito

Apresentamos a seguir as provas das propriedades *strong completeness* e *strong accuracy* do detector de defeitos. Estas provas levam em consideração que o detector executa em Spa, e que a propriedade *Sincronia Particionada Forte* é satisfeita. Também assumimos a existência do QoS Provider e do Detector de Estados dos Processos, e que este Detector de Estados satisfaz

as propriedades *strong completeness* e *strong accuracy*, para detector de estado.

Teorema 3.1 *O detector de defeitos apresenta a Propriedade Strong Completeness - Todas as falhas de processos serão detectadas permanentemente por todos os processos corretos em um tempo finito.*

Prova

Esta prova é construída por contradição, supondo existir um processo, p_x, que falha no tempo t, e cuja falha não será detectada por algum processo correto. Conseqüentemente, supõem-se a existência de um processo correto, p_y, que não irá detectar a falha do processo p_x. É assumido que $t' > t$ é o momento em que o processo p_y já recebeu a última mensagem $I - am - alive(p_x)$ enviada por p_x (após falhar de forma silenciosa p_x interrompe o envio de mensagens), e no qual o valor do relógio local de p_y se torna superior ao *timeout* calculado para a recepção da próxima mensagem $I - am - alive(p_x)$, ou seja, o momento em que p_y detecta a ausência de uma mensagem $I - am - alive(p_x)$.

No tempo t', p_y inicia a execução da tarefa 2 do algoritmo de detecção, uma vez que a condição $CT_y() > timeout_y[p_x]$ é satisfeita, e a partir deste tempo p_y não receberá mensagens $I - am - alive(p_x)$ do processo p_x. De acordo com a propriedade *Sincronia Particionada Forte*, temos que todos os processos do sistema, incluindo p_x, pertencem a alguma Componente Síncrona, e possuem canais de comunicação isócronos, tendo a sua identificação previamente inserida no conjunto $live_y$. Portanto, o if da linha 5 é satisfeito. Existem duas possibilidades de execução dependendo de o canal $c_{x/y}$ ser ou não isócrono.

1. $c_{x/y}$ é isócrono no tempo t'.
   - Ao executar o algoritmo de detecção no tempo t', o comando if da linha 6 é satisfeito, e a identificação de p_x é transferida do conjunto $live_y$ para o conjunto $down_y$ (linhas 7 e 8). Uma mensagem $notification(p_y, p_x)$ é enviada para todos os processos. Conseqüentemente, a partir do tempo t', p_y detecta a falha de p_x permanentemente.
2. $c_{x/y}$ não é isócrono no momento t'.
   - Como p_x possui ao menos um canal isócrono e o canal $c_{x/y}$ não é isócrono, existe um canal de comunicação $c_{x/z}$ que é isócrono, ligando p_x ao processo p_z. O processo p_z detecta a falha do processo p_x ao executar a tarefa 2 do algoritmo de detecção, quando a condição $CT_z() > timeout_z[p_x]$ é satisfeita (como descrito no item anterior desta prova). p_z transfere a identificação de p_x do conjunto $live_z$ para o conjunto $down_z$, e envia mensagens $notification(p_z, p_x)$ para todos os processos do sistema. Como os canais de comunicação são confiáveis, p_y recebe a mensagem $notification(p_z, p_x)$, informando a falha de p_x.

Como mostrado em todas as situações possíveis, o processo p_y detecta a falha do processo p_x permanentemente, em um tempo finito de sua execução, o que contradiz a suposição inicial da prova, provando o teorema. ∎

Teorema 3.2 *O detector de defeitos apresenta a propriedade Strong Accuracy - Nenhum processo será suspeito erroneamente por nenhum outro processo.*

Prova

Esta prova é desenvolvida por contradição, assumindo a existência de um processo correto p_x, e de um processo correto p_y, que irá continuamente suspeitar de p_x.

De acordo com a propriedade *Sincronia Particionada Forte*, todos os processos, inclusive p_x, fazem parte de alguma Componente Síncrona, tendo portanto a sua identificação inserida, previamente, nos conjuntos $live$ de todos os processos, inclusive $live_y$. Como, segundo a Propriedade *Strong Accuracy* do detector de estados dos processos, um processo não terá o seu estado erroneamente identificado, temos que a identificação de p_x permanece no conjunto $live_y$.

Temos as seguintes possibilidades de detecção:

1. Suspeita (linha 13 do algoritmo de detecção) - Como a condição $p_x \in uncertain_y$ não é satisfeita, a identificação do processo p_x não poderá ser inserida no conjunto $suspected_y$ (não satisfaz a condição da linha 12 do algoritmo de detecção), e o processo p_x não poderá ser erroneamente suspeito.
2. Notificação através do canal $c_{x/y}$ (linhas 6 a 9 do algoritmo de detecção) - Se $c_{x/y}$ for isócrono, todas as mensagens transferidas através deste canal serão recebidas dentro dos limites de tempo determinados. Neste caso a condição da linha 6 não será satisfeita, e p_x não poderá ser erroneamente detectado.
3. Notificação através da recepção da mensagem $notification(p_z, p_x)$ (linhas 17 a 19 do algoritmo de detecção) - Para que p_y receba uma mensagem *notification*(p_z, p_x), esta mensagem deve ter sido enviada por um processo p_z. Para enviar esta mensagem, p_z teria de executar as linhas 6 a 9 do algoritmo de detecção, e o canal $c_{z/x}$ tem de ser isócrono. Neste caso, como as mensagens transferidas através do canal $c_{z/x}$ são entregues dentro dos *timeouts* determinados (canais de comunicação isócronos fornecem comunicação com limites de tempo conhecidos e garantidos), e como p_x não falhou e continua a enviar mensagens, obtemos para p_z a situação do caso anterior. Neste caso, a condição da tarefa 2 do algoritmo de detecção, executado por p_z, não é satisfeita, e p_z não envia mensagens $notification(p_z, p_x)$ para nenhum processo. Como p_y não recebe nenhuma mensagem $notification(p_z, p_x)$, não irá executar as linhas 17 a 19 do algoritmo de detecção. p_x não será erroneamente detectado.

p_x não é detectado por p_y em nenhuma situação possível, o que contradiz a suposição inicial da prova. ∎

3.4. Detector Parcialmente Perfeito

Definimos um detector de defeitos como sendo da classe xP quando satisfizer a propriedade *strong completeness* e a nova propriedade *partially strong accuracy*. Esta propriedade determina que: nenhum processo que pertença a uma Componente Síncrona será suspeito erroneamente por nenhum outro processo. Este detector de defeitos é implementado pelo mesmo algoritmo descrito na seção anterior, ao executar sobre Spa, sendo satisfeita a propriedade *Sincronia Particionada Fraca*.

Sincronia Particionada Fraca: O conjunto dos processos que pertencem a componentes síncronos é menor que Π. Mais formalmente, seja $\Sigma = Cs_1 \cup Cs_2 \cup ... \cup Cs_k$, $k \geq 1$, o conjunto união de todos os possíveis componentes síncronos em DS. Então, $\Pi - \Sigma \neq \emptyset$. Ou seja, $\exists p_i \in \Pi$, tal que $p_i \notin \Sigma$ e $2 \leq |\Sigma| < |\Pi|$.

Existem, portanto, processos que pertencem a Componentes Síncronas, mas nem todo processo em Π pertence a alguma Componente Síncrona. Conseqüentemente, temos que: $\forall p_i \in \Pi, live_i \neq \oslash \wedge live_i \neq \Pi$, ou seja, existem processos identificados no estado $live$, e existem processos identificados no estado $uncertain$.

A propriedade *strong completeness* é provada de forma similar ao apresentado na seção anterior. A seguir apresentamos a prova formal da propriedade *partially strong accuracy*, a partir de Spa e da propriedade Sincronia Particionada Fraca, além das informações fornecidas pelo QoS Provider, e das propriedades *strong completeness* e *strong accuracy* do Detector de Estados dos Processos.

Teorema 3.3 *O detector de defeitos apresenta a propriedade partially Strong Accuracy - Nenhum processo que pertença a uma Componente Síncrona será suspeito erroneamente por nenhum outro processo.*

Prova

Esta prova é desenvolvida por contradição, assumindo a existência de um processo correto p_x, que pertence a uma Componente Síncrona, e de um processo correto p_y, que irá continuamente suspeitar de p_x.

Como o processo p_x pertence a uma Componente Síncrona, ele possui ao menos um canal de comunicação isócrono, interligando p_x a algum outro processo. Segundo a propriedade *Strong Completeness* do detector de estado, todo processo que possui ao menos um canal de comunicação isócrono é identificado no estado $live$, e terá sua identificação inserida no conjunto $live_i$ para todo processo p_i de Π, portanto, $p_x \in live_y$. Como, segundo a Propriedade *Strong Accuracy* deste mesmo detector, um processo não terá o seu estado erroneamente identificado, temos que a identificação de p_x permanece no conjunto $live_y$.

Temos as seguintes possibilidades de detecção:

1. Suspeita (linha 13 do algoritmo de detecção) - Como a condição $p_x \in uncertain_y$ não é satisfeita, a identificação do processo p_x não poderá ser inserida no conjunto $suspected_y$ (não satisfaz a condição da linha 12 do algoritmo de detecção), e o processo p_x não poderá ser erroneamente suspeito.
2. Notificação através do canal $c_{x/y}$ (linhas 7 a 9 do algoritmo de detecção) - Se o canal $c_{x/y}$ for isócrono, todas as mensagens transferidas através deste canal serão recebidas dentro dos limites de tempo determinados. Neste caso a condição da tarefa 2 não será satisfeita, e p_x não poderá ser erroneamente detectado.
3. Notificação através da recepção da mensagem $notification(p_z, p_x)$ (linhas 17 a 19 do algoritmo de detecção) - Para que p_y receba uma mensagem *notification*(p_z, p_x), esta mensagem deve ter sido enviada por um processo p_z. Para enviar esta mensagem, p_z teria de executar as linhas 6 a 9 do algoritmo de detecção, e o canal $c_{z/x}$ tem de ser isócrono. Neste caso, como as mensagens transferidas através do canal $c_{z/x}$ são entregues dentro dos *timeouts* determinados (definição dos canais isócronos), e como p_x não falhou e continua a enviar mensagens, obtemos para p_z a situação do caso anterior. Neste caso, a condição da tarefa 2 do algoritmo de detecção, executado por p_z, não é satisfeita, e p_z não envia mensagens $notification(p_z, p_x)$ para nenhum processo. Como p_y não recebe nenhuma mensagem $notification(p_z, p_x)$, não irá executar as linhas 17 a 19 do algoritmo de detecção. p_x não será erroneamente detectado.

p_x não é detectado por p_y em nenhuma situação possível, o que contradiz a suposição inicial da prova. ∎

4. Conclusões

A incapacidade em detectar defeitos de forma precisa está no cerne da impossibilidade de resolver problemas de consenso em sistemas assíncronos. Portanto, acreditava-se que um sistema assíncrono equipado com detector de faltas perfeito (P) se equivaleria a um sistema síncrono (onde se pode facilmente implementar detectores perfeitos). No entanto, existia uma crença estabelecida de que detectores perfeitos de defeitos somente seriam implementáveis em sistemas síncronos. No presente artigo desmontamos essa crença, mostrando como um detector perfeito de defeitos pode ser implementado num sistema mais fraco que o sistema distribuído síncrono. Para isso, introduzimos o sistema síncrono particionado (Spa) que é estritamente mais fraco que o sistema síncrono, haja vista que em um sistema Spa processos podem estar conectados por canais sem garantias temporais - tornando impossível a implementação de ações síncronas globais como sincronização interna de relógios. Através da propriedade que definimos como *Sincronia Particionada Forte*, mostramos como implementar P em Spa. Melhor ainda, mostramos que mesmo que *Sincronia Particionada Forte* não possa ser garantida, podemos ainda assim tirar proveito das partições síncronas existentes para melhorar a robustez das aplicações de tolerância a falhas, através de um detector parcialmente perfeito, denominado por nós de xP. Um algoritmo de consenso adaptativo que tira proveito desse tipo de detector foi apresentado em [Gorender et al. 2007]. Sistemas Spa se adéquam em especial a configurações de aglomerados (*clusters*) interligados por redes de longa distância (como a Internet). Nessas circunstâncias, processos de um *cluster* formam um componente síncrono e a redistribuição de carga, por exemplo, devido a falha de processos, pode ser feita de forma mais eficiente uma vez que processos são detectados falhos de forma não ambígua. As propriedades e algoritmos necessários para implementar P e xP foram introduzidos no artigo, assim como as provas de correção relacionadas.

Referências

Aurrecoechea, C., Campbell, A. T., and Hauw, L. (1998). A survey of qos architectures. *ACM Multimedia Systems Journal, Special Issue on QoS Architecture*, 6(3):138–151.

Chandra, T. D. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267.

Charron-Bost, B., Guerraoui, R., and Schiper, A. (2000). Synchronous system and perfect failure detector: solvability and efficiency issues. In *Proceedings of the International Conference on Dependable System and Networks*, pages 523–532.

Chen, W., Toueg, S., and Aguilera, M. K. (2000). On the quality of service of failure detectors. In *Proceedings of the International Conference on Dependable Systems and Networks (ICDSN/FTCS-30)*, pages 561–580.

Cristian, F. and Fetzer, C. (1999). The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657.

Dwork, C., Lynch, N., and Stockmeyer, L. (1988). Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323.

Falai, L. and Bondavalli, A. (2005). Experimental evaluation of the qos of failure detectors on wide area network. In *Proceedings of the International Conference on Dependable Systems and Networks, DSN*, pages 624–633.

Fetzer, C. and Cristian, F. (1996). Fail-awareness in timed asynchronous systems. In *Proceedings of the 15th Symposium on Principles of Distributed Computing*, pages 314–321.

Fisher, M. J., Lynch, N., and Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382.

Gorender, S. (2005). Um modelo híbrido e adaptativo para sistemas distribuídos tolerantes a falhas. Tese de Doutorado aprovada pelo CIN/UFPE.

Gorender, S., Macêdo, R., and Raynal, M. (2007]). An adaptive programming model for fault-tolerant distributed computing. *IEEE Transactions on Dependable and Secure Computing*, 4(1):18–31.

Gorender, S. and Macêdo, R. J. d. A. (2002). Um modelo para tolerância a falhas em sistemas distribuídos com qos. In *Anais do Simpósio Brasileiro de Redes de Computadores, SBRC 2002*, pages 277–292.

Lamport, L., Shostak, R., and Pease, M. (1982). The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401.

Lima, F. R. L. and Macêdo, R. J. d. A. (2005). Adapting failure detectors to communication network load fluctuations using snmp and artificial neural nets. In *Second Latin-American Symposium on Dependable Computing (LADC2005), Lecture Notes in Computer Science*, pages 191 – 205.

Lynch, N. A. (1996). *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc.

Macêdo, R. (2000). Failure detection in asynchronous distributed systems. In *Proc. of II Workshop on Tests and Fault-Tolerance (II WFT 2000)*, pages 76–81.

Macêdo, R., Gorender, S., and Cunha, P. (2005). The implementation of a qos-based adaptive distributed system model for fault tolerance. In *Anais do Simposio Brasileiro de Redes de Computadores (SBRC 2005)*, pages 827–840.

Mullender, S. (1993). *Distributed Systems*. Addison-Wesley Pub Co.

Nunes, R. C. and Jansch-Pôrto, I. (2004). Qos of time-out based self-tuned failure detector: the effects of its communication delay predictor and its safety margin. In *IEEE Int. Conf. on Dependable System and Network, track on Performance and Dependability Symposium (DSN-PDS)*, pages 753–761.

Schiper, A. (1997). Early consensus in an asynchronous systems with a weak failure detector. *Distributed Computing*, 10(3):149–157.

Veríssimo, P. and Casimiro, A. (2002). The timely computing base model and architecture. *IEEE Transactions on Computers*, 51(8):916–930.

Abordagem para Desviar Pacotes na Presença de Falha em Backbones IP com OSPF

Fernando Barreto[1], Emilio C. G. Wille[1], Luiz Nacamura Junior[1]

[1]Pós-Graduação em Engenharia Elétrica e Informática Industrial – Universidade Tecnológica Federal do Paraná (UTFPR)
CEP – 80230-901 – Curitiba – PR – Brazil

{fbarreto,ewille,nacamura}@utfpr.edu.br

Abstract. *IP network backbones use link state routing protocols to find correct routes. In face of a topology change, e.g. a failure, these protocols need some time to react to it in order to find new routes. During this time, the routes become unstable, causing high packet loss rate and depreciation of backbone reliability. This work presents a proactive approach to help the IP routing protocols during the convergence period in order to reduce packet loss rate. The approach is evaluated using various artificial and real topologies, and a simulation is also implemented in order to evaluate the packet loss rate reduction, which yielded satisfactory results.*

Resumo. *Os backbones IP utilizam protocolos de roteamento do tipo estado do enlace para definir as rotas corretamente. Em situações de mudança na topologia, como uma falha, esses protocolos necessitam de um tempo para reagir e encontrar novas rotas. Durante esse tempo, as rotas ficam instáveis com alta taxa de pacotes perdidos e queda na confiabilidade do backbone. Esse trabalho propõe uma abordagem pró-ativa para auxiliar os protocolos de roteamento a reduzir a taxa de pacotes perdidos durante esse período. Essa abordagem é avaliada em representações de topologias artificiais e reais, e também em simulação para analisar a redução de pacotes perdidos, demonstrando resultados bastante satisfatórios.*

1. Introdução

As infra-estruturas de redes IP atuais são utilizadas como ambiente de transporte para tráfego de informações em geral. Devido ao seu uso genérico e crescente, tais infra-estruturas se tornam críticas e necessitam de alta confiabilidade, pois a interrupção de serviço causada por uma falha pode comprometer o desempenho dos tráfegos em geral. Dentre os cenários de falha existentes, as falhas transitórias e de um único componente de rede são as de maior ocorrência em *backbones* IP [Iannaccone et al. 2004] [Nelakuditi et al. 2007]. Entende-se por falha de um único componente de rede como sendo a falha de um enlace, roteador ou *Shared Risk Link Group* (SRLG). Um SRLG pode ser considerado um único componente, pois é constituído de um conjunto de enlaces dependentes que pertencem a um mesmo meio comum, e.g. duto com várias fibras, onde a falha de um enlace deve implicar na falha dos demais enlaces dependentes.

O processo de recuperação de uma falha deve ser o mais rápido possível para reduzir significativamente o comprometimento dos fluxos de tráfego passantes. O objetivo é manter

na medida do possível o nível de confiabilidade da infra-estrutura de rede. Dentre as abordagens utilizadas para esse processo de recuperação tem-se a recuperação em nível de rede e em nível de *hardware*. Em nível de rede tem-se a recuperação por recálculo de rotas [Iannaccone et al. 2004] e a proteção por rotas alternativas pré-calculadas [Shand and Bryant 2007]. Já em nível de *hardware* tem-se a instalação e distribuição de caminhos físicos de proteção (mais eficiente, porém muito mais cara).

O processo de recuperação por recálculo de rotas é um processo reativo executado pelos protocolos de roteamento atuais do tipo estado do enlace, como o OSPF [Moy 1998]. Para que o OSPF realize esse processo, ele deve reconhecer a falha, notificar a ocorrência aos roteadores vizinhos (que por sua vez divulgam para os seus vizinhos até cobrir toda a topologia), esperar um tempo necessário para receber todas as notificações dos outros roteadores, calcular os menores caminhos (SPF – algoritmo de Dijkstra) com as novas informações do estado do enlace, para então atualizar a tabela de encaminhamento (*Forwarding Information Base* – FIB). Todo esse processo, nomeado período de convergência, atinge em torno de dezenas de segundos [Iannaccone et al. 2004], o que compromete seriamente a confiabilidade da infra-estrutura de rede. Para reduzir esse período, [Francois et al. 2005] modificou o tempo de cada item integrante desse processo tornando possível reduzir esse período para a casa de décimos de segundo. Porém, essa redução pode causar mais instabilidades [Basu and Riecke 2001] sendo dependente da topologia, da velocidade de processamento dos roteadores e dos enlaces.

As rotas alternativas pré-calculadas são abordagens que seguem o *framework* de *IP Fast ReRouting* (IPFRR) [Shand and Bryant 2007]. Esse *framework* define um esquema teórico para realizar um processo de recuperação local com caminhos previamente calculados para contornar uma falha. Dentre as principais abordagens IPFRR tem-se: FIR [Nelakuditi et al. 2007], NotVia [Bryant et al. 2007], LFA [Atlas and Zinin 2007], Tunnels [Bryan et al. 2005] e MRC [Kvalbein et al. 2006]. Entretanto, apenas o NotVia consegue atingir 100% de cobertura para falha de um enlace, roteador ou SRLG. O NotVia distribui endereços *not-via* na topologia para representar cada componente de rede a ser contornado. Uma rota para um endereço *not-via* em todos os roteadores consegue definir um caminho que contorna o componente que ele representa. Entretanto, o NotVia pode gerar caminhos mais extensos que o necessário para desviar os pacotes devido à localização dos endereços *not-via* (*next-next-hop*), além de ocupar recursos excessivos na FIB para representá-los e depender da técnica de encapsulamento (como conseqüência pode fragmentar ou até descartar os pacotes). Maiores detalhes sobre o NotVia e as demais abordagens não são descritos por restrição de espaço.

Este trabalho propõe uma nova abordagem formalizada e completa da proposta inicial do *Esquema de Caminhos Emergenciais Rápidos* (E-CER) [Barreto et al. 2007], incluindo a extensão na FIB. Essa abordagem adapta a idéia do *framework* IPFRR para fornecer caminhos mais curtos que as abordagens IPFRR e, dependendo da falha, sempre iguais aos obtidos pelo processo OSPF após sua convergência (permite uma adaptação gradual às novas rotas OSPF). Além disso, o E-CER necessita de menos recursos para representar esses caminhos na FIB e independe da técnica de encapsulamento.

2. E-CER

E-CER é uma abordagem distribuída baseada no *framework* IPFRR para calcular os

caminhos de recuperação, cada um nomeado *Caminho Emergencial Rápido* (CER). Esses caminhos são disponibilizados em forma de marcas (*CER_Marca/IR*) na FIB, que são utilizadas por um processo auxiliar de encaminhamento nomeado *CER_EncDif*.

Para uma abordagem de recuperação de falha cobrir 100% da topologia, esta deve possuir uma infra-estrutura física capaz de manter a conectividade entre todos os roteadores na presença de uma falha de enlace, roteador ou SRLG. Além disso, uma distribuição do tráfego de rede na topologia deve ser planejada para ocupar, no máximo, 50% da capacidade dos enlaces. Este planejamento já é realizado em várias topologias reais para permitir a acomodação de tráfego desviado na presença de falhas [Iannaccone et al. 2004]. Todas essas restrições podem ser adotadas por qualquer abordagem IPFRR, pois caso contrário, não será possível alcançar 100% de cobertura de falha.

2.1. Cálculo dos CERs

O E-CER considera a métrica OSPF existente (soma dos custos dos enlaces [Moy 1998]) e o número de roteadores na geração dos CERs. O uso da métrica do OSPF permite o reuso das rotas já existentes, o que reduz a complexidade do cálculo desses caminhos. Este trabalho adapta outras abordagens IPFRR para melhor utilizá-las, suporta SRLG, pesos de enlaces assimétricos, independe da técnica de encapsulamento e atinge 100% de cobertura de falha.

Cada roteador gera seus próprios CERs e este cálculo deve seguir a execução do processo OSPF, pois tem por objetivo reusar a base de dados do estado do enlace atualizada com a configuração da topologia.

Considere uma topologia de rede representada em um grafo G(*V*,*A*), onde *V* é o conjunto de roteadores e *A* o conjunto de enlaces que conecta os roteadores. Considere $(i, j) \in A$ a representação de um enlace conectando os roteadores *i* e *j*. Para cada (i, j), um custo $c_{i,j}$ é especificado durante a configuração do OSPF.

O cálculo SPF executado por um *Roteador Origem* (*RO*) basicamente minimiza a soma dos custos dos enlaces $c_{i,j}$ de *RO* até qualquer outro roteador na topologia. A primeira formulação matemática para cálculo dos CERs é uma adaptação da formulação de programação linear para o *Shortest Path Problem* [Ahuja et al. 1993]. O cálculo localiza a função custo $Z_{RO,J,RA}$ que minimiza a soma dos custos dos enlaces respeitando o número de unidades de fluxo que atravessam os enlaces para descobrir os caminhos de *RO* até um conjunto de roteadores destino *J*, onde $RO \notin J$, e impede o uso dos componentes adjacentes ao *RO*: *Roteador Adjacente* (*RA*) ou *Enlace Adjacente (RO,RA)*.

A restrição *(a)* seleciona os menores caminhos transmitindo *n-1* unidades de fluxo de *RO*, significando que a cada roteador alcançado no menor caminho 1 unidade de fluxo é consumida. As restrições *(b)* e *(c)* impedem os caminhos que utilizam *(RO,RA)*, se *J=RA*, ou *RA* (todos os enlaces adjacentes a *RA*), se *J≠RA*, pois define um custo de enlace infinito. Dessa forma, a formulação procura sempre encontrar caminhos que contornam *RA* quando possível, pois *RO*, na presença de falha, não consegue identificar imediatamente se uma falha é do *(RO,RA)* ou do *RA* e o contorno do *RA* automaticamente propicia o contorno do *(RO,RA)*. Se existem enlaces pertencentes a um SRLG que contém *(RO,RA), se J=RA*, ou *(RA,?)*, se *J≠RA*, a restrição *(d)* atribui pesos infinitos a esses enlaces para serem evitados na escolha do menor caminho.

$$Z_{RO,J,RA} = \min \sum_{i=1}^{n} \sum_{j=1}^{n} c_{i,j} x_{i,j}$$

Sujeito a:

$$\sum_{j=1}^{n} x_{i,j} - \sum_{k=1}^{n} x_{k,i} = \begin{cases} n\text{-}1, & if\ i = 1 \\ -1, & if\ i \neq 1 \end{cases} \quad (a)$$

$$c_{RO,j} = \infty \ \ se\ \ j = RA\ e\ se\ \ J = RA \quad (b)$$

$$c_{i,j} = \infty \ \ se\ j = RA\ e\ se\ \ J \neq RA \quad (c)$$

$$c_{i,j} = \infty \ \ se\ (i,j) \in \begin{cases} SRLG(RO,RA)\ ,\ se\ J = RA \\ SRLG(RA,?),\ se\ J \neq RA \end{cases} \quad (d)$$

$$i > 0\ ;\ j > 0\ ;\ i \neq j\ \ ;\ \ x_{i,j} \geq 0\ ;\ c_{i,j} \geq 0 \quad (e)$$

A solução para esse problema pode ser obtida com uma modificação simples no algoritmo de Dijkstra (SPF) para considerar as restrições *(b)*, *(c)* e *(d)*. Como apenas parte da árvore SPF é afetada pelo contorno do componente, o tempo de recálculo dos menores caminhos com *Incremental-SPF* [Narvaez 2000] é reduzido consideravelmente.

Os menores caminhos que obedecem a $Z_{RO,J,RA}$ são representados em um conjunto $\varphi_{RO,J,RA}$. Considere os menores caminhos para um *Roteador Destino* (*RD*) que pertencem ao $\varphi_{RO,J,RA}$, onde $RD \in J$. Se *RD* tem o seu menor caminho original OSPF (com todos os componentes na topologia) de *RO* até *RD* utilizando *(RO,RA)*, então os menores caminhos obtidos com $Z_{RO,J,RA}$ são *caminhos alternativos* para *RD* e são candidatos para contornar *(RO,RA)* ou *RA* dependendo se *RD=RA*. O conjunto de possíveis caminhos alternativos para *RD* que possui essa característica são representados como $\varphi_{RO,RD,RA}$. Cada caminho alternativo pertencente a $\varphi_{RO,RD,RA}$ é representado como $ALT_{RO,RD,RA}$.

A segunda formulação matemática abaixo ($Z'_{RO,RD,RA}$) foi criada para identificar, em cada $ALT_{RO,RD,RA}$, qual roteador é capaz de contornar uma falha de componente, i.e. *(RO,RA)* ou *RA*, quando, desse roteador, o caminho original OSPF pode seguramente encaminhar os pacotes pelo IP de destino até *RD*. O processo de identificação adapta a proposta do IPFRR [Shand and Bryant 2007] para criar uma classificação de níveis (ECMP, LFA e SIG) e facilitar a localização desse roteador, que recebe o nome de *Roteador CER* (*RC*). Cada roteador de $ALT_{RO,RD,RA}$ que é analisado na formulação para ser *RC* é nomeado *k*. O nível ECMP identifica o *RC* no primeiro roteador (*Roteador Vizinho – RV*) de $ALT_{RO,RD,RA}$ e o $ALT_{RO,RD,RA}$ deve ter o mesmo custo OSPF (soma dos custos dos enlaces, i.e. *DistOSPF*) que o caminho original OSPF de *RO* até *RD*. O nível LFA adapta a abordagem LFA original [Atlas and Zinin 2007] para identificar *RC* no *RV* e ainda considera a restrição do número de roteadores no caminho alternativo. O nível SIG é uma extensão da abordagem LFA para identificar o *RC* após o *RV*, dessa forma *k* segue através da seqüência de roteadores de $ALT_{RO,RD,RA}$ até encontrar um roteador capaz de ser o *RC*. Todos os *k* que não são capazes de ser *RC* são definidos como *roteadores_intermediários*. Portanto, independente do nível utilizado, um sub-caminho é gerado para cada $ALT_{RO,RD,RA}$ e recebe a identificação de $CER_{RO,RD,RA}$. Um $CER_{RO,RD,RA}$ é uma seqüência de roteadores até *RC*: se nível ECMP ou LFA então a seqüência é somente {*RC*} com *RC=RV*, já no nível SIG a seqüência é {*RV*, *roteadores_intermediários, RC*}. Todos os $CER_{RO,RD,RA}$ gerados para $\varphi_{RO,RD,RA}$ são candidatos ao CER selecionado para o *RD*, nomeado $S\_CER_{RO,RD,RA}$.

$$Z'_{RO,RD,RA} = \min(custo\_CER_{RO,RD,RA}\ num\_CER_{RO,RD,RA})$$

Sujeito a:

$$\varphi_{RO,RD,RA} \subset \varphi_{RO,J,RA} \quad \text{(a)}$$

$$ALT_{RO,RD,RA} \in \varphi_{RO,RD,RA} \quad \text{(b)}$$

$$CER_{RO,RD,RA} \subset ALT_{RO,RD,RA} \quad \text{(c)}$$

$$RC = k,\ se\ DistOSPF(RO,k)+DistOSPF(k,RD)=DistOSPF(RO,RA)+DistOSPF(RA,RD),\ e\ \ k \in Vizinhos\_RO \quad \text{(d)}$$

$$RC = k,\ se\ DistOSPF(k,RD)<DistOSPF(k,RO)+DistOSPF(RO,RD),\ e\ \ k \in Vizinhos\_RO \quad \text{(e)}$$

$$RC = k,\ se\ DistOSPF(k,RD)<DistOSPF(k,RO)+DistOSPF(RO,RD),\ e\ \ k \notin Vizinhos\_RO \quad \text{(f)}$$

$$RA \notin OSPF\_Roteadores(k,RD),\ se\ RA \neq RD \quad \text{(g)}$$

$$OSPF\_Enlaces(k,RD) \cap \begin{cases} SRLG(RO,RA)=\varnothing,\ se\ RA=RD \\ SRLG(RA,?)=\varnothing,\ se\ RA \neq RD \end{cases} \quad \text{(h)}$$

$$k \in CER_{RO,RD,RA},\ se\ nenhuma\ restrição\ (d),\ (e)\ ou\ (f)\ for\ possível \quad \text{(i)}$$

$$k \in CER_{RO,RD,RA}\ e\ k\ é\ o\ último\ roteador,\ se\ (RC=k)\ e\ (uma\ restrição\ (d),(e)\ ou\ (f)\ for\ possível) \quad \text{(j)}$$

$$num\_CER_{RO,RD,RA} = \begin{cases} 1\ unidade,\ se\ restrição\ (d)\ for\ possível \\ NumRoteadoresOSPF(k,RD),\ se\ restrição\ (e)\ for\ possível \\ NumRoteadoresCER_{RO,RD,RA},\ se\ restrição\ (f)\ for\ possível \end{cases}, \quad se\ (j)\ for\ possível \quad \text{(k)}$$

$$custo\_CER_{RO,RD,RA} = \begin{cases} DistOSPF(k,RD),\ se\ restrição\ (d)\ ou\ (e)\ for\ possível \\ DistCER_{RO,RD,RA},\ se\ restrição\ (f)\ for\ possível \end{cases}, \quad se\ (j)\ for\ possível \quad \text{(l)}$$

$$k \in ALT_{RO,RD,RA}\ ;\ RA \notin Vizinhos\_RO \quad \text{(m)}$$

Para identificar qual o $S\_CER_{RO,RD,RA}$ para um *RD*, a função custo $Z'_{RO,RD,RA}$ minimiza o valor do $custo\_CER_{RO,RD,RA}$ com o número de roteadores $num\_CER_{RO,RD,RA}$. A localização do *RC* é obtida por análise de cada roteador na seqüência de roteadores de $ALT_{RO,RD,RA}$, i.e. variável *k*. Esta variável torna-se *RC*, o que indica o último roteador do $CER_{RO,RD,RA}$ na restrição *(j)*, se uma das restrições na seqüência for aceita: *(d)* para nível ECMP, *(e)* para nível LFA ou *(f)* para nível SIG. No nível SIG, *k* não pertence ao conjunto dos roteadores vizinhos a *RO* (*Vizinhos_RO*), pois *k* sempre é um roteador após o *RV* em $ALT_{RO,RD,RA}$. Se nenhuma dessas restrições for aceita, então *k* pertence ao $CER_{RO,RD,RA}$, i.e. restrição *(i)*, e outro *k* na seqüência de $ALT_{RO,RD,RA}$ deve ser analisado. Restrição *(g)* impede a seleção de $ALT_{RO,RD,RA}$ se existir uma seqüência de roteadores no caminho OSPF de *k* até *RD* que utiliza *RA*, se *RA≠RD*. A restrição *(h)* impede a seleção de $ALT_{RO,RD,RA}$ se existir uma seqüência de enlaces no caminho OSPF de *k* até *RD* que utiliza algum enlace pertencente a um SRLG. Restrição *(k)* define um valor para $num\_CER_{RO,RD,RA}$ de acordo com o nível escolhido: 1 para o nível ECMP *(d)*, o número de roteadores no caminho OSPF, i.e. $NumRoteadoresOSPF$ de *k* até *RD*, para o nível LFA *(e),* ou o número de roteadores no caminho CER (*RV* até *RC*), i.e. $NumRoteadoresCER_{RO,RD,RA}$, para o nível SIG *(f)*. O nível ECMP atribui o valor 1 na restrição *(k)* com o objetivo de obter múltiplos caminhos de recuperação de mesmo custo para permitir balanceamento de carga com ECMP já adotado no OSPF. A restrição *(l)* também define o valor de $custo\_CER_{RO,RD,RA}$ de acordo com o nível escolhido: soma dos custos dos enlaces do caminho OSPF de *k* até *RD* (*DistOSPF*) para os níveis ECMP e LFA, ou a soma dos custos dos enlaces do caminho CER de *RV* até *RC*, i.e. $DistCER_{RO,RD,RA}$, para o nível SIG.

Portanto, a formulação obtém, para cada *RD*, o menor $S\_CER_{RO,RD,RA}$ de todos os caminhos alternativos de $\varphi_{RO,RD,RA}$ possíveis e com o menor número de roteadores.

Um algoritmo foi proposto para calcular $Z_{RO,J,RA}$ e $Z'_{RO,RD,RA}$. Obtém-se o conjunto $\varphi_{RO,RD,RA}$ e por fim identifica o $S\_CER_{RO,RD,RA}$ de forma distribuída (em cada roteador). O $S\_CER_{RO,RD,RA}$ selecionado para cada *RD* é armazenado em um vetor: *CER_Vetor*[*RD*].

O caminho obtido pelo E-CER é sempre o mesmo menor caminho obtido pelo OSPF após o término de convergência se a falha for do *RA*. Essa afirmação é provada pela geração pró-ativa dos CERs, que sempre contornam o *RA* quando possível (*RA≠RD*, pois contorna automaticamente *(RO,RA)*), mesmo se uma falha real for apenas do *(RO,RA)*. Com isso, o E-CER permite com sucesso uma adaptação gradual de todos ou de grande parte dos roteadores do CER (de *RV* ao *RC*) aos novos caminhos OSPF obtidos durante o período de convergência sem a alteração do sentido das rotas.

2.2. Extensão E-CER na FIB

Uma vez que os $S\_CER_{RO,RD,RA}$ para todos os *RD* foram gerados em cada roteador e armazenados em *CER_Vetor*[*RD*], o E-CER deve adicionar uma extensão na FIB para representar esses vetores. Essa extensão será utilizada imediatamente durante a ocorrência de falha. Todas as abordagens relacionadas também possuem uma extensão similar, mas no caso do E-CER, essa extensão consiste de um par utilizado na marcação de pacotes (*CER_Marca / Interface de Rede* (*IR*)). Esse par é utilizado por um processo de encaminhamento auxiliar ao encaminhamento IP padrão nomeado *CER_EncDif*.

2.2.1 Marcação *CER_Marca/IR*

Uma *CER_Marca* é a representação de marca para pacotes, que é definido por cada roteador (*RO*) durante o cálculo dos CERs e é relacionado a um *CER_Vetor*[*RD*] selecionado. Essa marca é representada com 16 bits divididos em 10 bits para identificar o *RO* (*RO_id*), e 6 bits para representar um *CER_Vetor*[*RD*] selecionado (*CER_id*). O *RO_id* possibilita no máximo 1024 roteadores IGP em uma mesma área OSPF, com no máximo 64 *CER_id* possíveis por roteador. A limitação de 1024 roteadores por área OSPF é mais que suficiente, pois, em situações práticas, uma área OSPF definida com mais de 1024 roteadores não é escalável devido ao alto tráfego de informações do estado do enlace. Os 6 bits para *CER_id* são suficientes, pois em nossos testes com representação de topologias artificiais e reais foram necessários no máximo 40 diferentes *CER_ids* em roteadores isolados e em média foram necessários 16 *CER_ids*.

Para representar os 16 bits *CER_Marca* no pacote IP, evitou-se utilizar mecanismos de encapsulamento tanto no cabeçalho IPv4 quanto no IPv6. Para tanto o E-CER procura reusar campos inutilizados desses cabeçalhos:

- IPv4: Os 16 bits do campo *Fragment Offset*, com a padronização da abordagem *Path MTU Discovery* (PMTU) [Mathis and Heffner 2007];
- IPv6: Usam-se os 16 últimos bits do campo *Flow Label* utilizando a definição dos três primeiros bits com "111" para uso futuro [Tang et al. 2002].

O *IR* é representado com 8 bits, possibilitando até no máximo 255 interfaces de rede por roteador, sendo suficiente para os *hardwares* de roteadores atuais.

Um par *CER_Marca/IR* poderia ser obtido para cada *CER_Vetor*[*RD*] gerado, porém uma seleção desses vetores é realizada para reduzir a quantidade de

CER_Marca/IR. Para gerar esses pares, deve-se inicialmente seguir a duas regras:

- Todas as entradas existentes na FIB (geradas pelo OSPF) com prefixos de rede anunciados por um mesmo *RD* (informação obtida da base de informações do estado do enlace) devem usar o mesmo *CER_Vetor*[*RD*]. Neste caso, todas estas entradas na FIB devem referenciar um único par *CER_Marca/IR* que representa esse vetor, pois todos os pacotes, quando desviados, deverão seguir o mesmo CER para *RD*;
- Se existir dois ou mais *CER_Vetor* com diferentes *RD*, porém com a mesma seqüência de roteadores, então todas as entradas na FIB com os prefixos anunciados por estes *RDs* devem referenciar um mesmo par *CER_Marca/IR*. Isto é possível uma vez que o CER a ser usado é o mesmo para os diferentes *RDs* necessitando apenas de uma marca;

Após observar essas duas regras, os pares *CER_Marca/IR* são gerados e adicionados na FIB seguindo um dos possíveis casos: *RO_caso* e *Não_RO_caso*.

O *RO_caso* ocorre quando um roteador gera seus próprios CERs, sendo esse roteador o *RO* no cálculo dos CERs. Nesse caso, para cada *CER_Vetor*[*RD*] selecionado uma *CER_Marca* é gerada contendo como *RO_id* os últimos 10 bits do endereço *loopback* do *RO* (*Router ID*) [Moy, 1998], o que é possível, pois os *Router Ids* são geralmente organizados em um mesmo grupo de endereços IP usado para os roteadores IGP. Se o *CER_Vetor*[*RD*] é obtido pelo nível ECMP ou LFA, então o *CER_id* contém 6 bits com valor "0". Caso contrário (nível SIG), o *CER_id* é gerado durante a execução do algoritmo para cálculo dos CERs através de um número identificador, que é incrementado a cada *CER_Vetor*[*RD*] de nível SIG selecionado. O *IR* a ser utilizado é a identificação da interface do *RO* conectada ao primeiro roteador do *CER_Vetor*[*RD*], i.e. *RV*.

O *Não_RO_caso* ocorre quando um roteador pertence à seqüência de roteadores de um *CER_Vetor*[*RD*] de nível SIG gerado por outro roteador. Neste caso, um roteador pertencente ao *Não_RO_caso* pode ser o *RV, roteadores_intermediários,* ou *RC*. Portanto, o par *CER_Marca/IR* é gerado utilizando o *CER_Vetor*[*RD*] e a *CER_Marca* obtidos a partir do processo *CER-EXT*, que é detalhado na seção 2.2.3. Com esses dados, cada roteador pertencente ao *Não_RO_caso* (*RV, roteadores_intermediários, RC*) somente necessita identificar o *IR* para gerar o par *CER_Marca/IR*. Para tanto, há de se considerar um roteador Y, como sendo um dos roteadores *Não_RO_caso*, esse roteador consegue facilmente identificar *IR* como sendo a interface de Y conectada ao próximo roteador após Y na seqüencia de roteadores de *CER_Vetor*[*RD*].

Com os pares *CER_Marca/IR* gerados, a Figura 1 ilustra como estes pares são representados na FIB. Existem três pares referenciados pelas entradas existentes na FIB, mas para simplificar o exemplo, somente duas entradas são representadas com os endereços de destino anunciados pelo mesmo roteador (*D*).

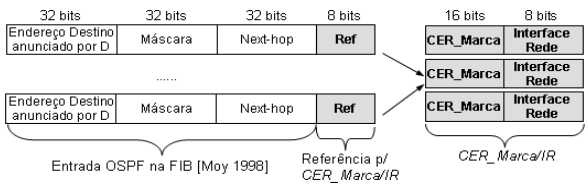


Figura 1. Representação da *CER_Marca/IR* na FIB

Estas duas entradas referenciam um mesmo par *CER_Marca/IR*, o que reduz o número necessário de marcas. A referência para esses pares é obtida através de um campo adicional *Ref* de 8 bits, o qual é suficiente para referenciar todos os *CER_Marca/IR* gerados pelo *RO_caso* e *Não_RO_caso.* Em nossos experimentos com representações de topologias artificiais e reais, apenas em alguns casos de roteadores isolados há a necessidade de no máximo de 6 bits para *Ref*, mas em média são necessários apenas 4 bits.

É importante salientar que somente os prefixos de rede anunciados pelo OSPF adicionam essa referência *Ref,* pois os demais prefixos de rede que são anunciados pelo *Border Gateway Protocol* (BGP) no interior de uma área reutilizam os prefixos definidos pelo OSPF para alcançar o *next-hop* BGP. Em caso de múltiplas áreas OSPF, o E-CER utiliza apenas a base de informações do estado do enlace de sua área, e os roteadores do tipo *Area Border Gateway* [Moy 1998] anunciam um sumário dos prefixos de rede de outras áreas. Dessa forma esses roteadores agem como *RD* na formulação sendo os anunciadores desses prefixos de rede na sua área.

Com os pares *CER_Marca/IR* definidos e preenchidos na FIB, o processo de encaminhamento *CER_EncDif* é capaz de utilizá-los na presença de falha.

2.2.2 Processo de Encaminhamento *CER_EncDif*

O processo de encaminhamento *CER_EncDif* tem por objetivo desviar os pacotes para que atinjam *RC*, e a partir desse roteador, o encaminhamento padrão baseado no endereço IP de destino é reusado pelo *CER_EncDif* para que os pacotes atinjam *RD*.

Na presença de uma falha adjacente a um roteador (torna-se o *RO*), o processo *CER_EncDif* é acionado e assume o encaminhamento para iniciar o desvio dos pacotes. O *CER_EncDif* identifica qual o *CER_Marca/IR* pela *Ref* na entrada da FIB a ser utilizada e realiza a marcação dos pacotes encaminhando-os para a respectiva *IR*. Se não existir a marca, então a topologia não possibilita 100% de cobertura de falha (seção 2).

A marcação é realizada diferentemente no IPv4 e no IPv6. No caso do IPv4, o *CER_EncDif* utiliza um bit do cabeçalho para diferenciar se o campo *Fragment Offset* está sendo usado pela marca ou pelo procedimento IPv4 de fragmentação de pacotes. O bit verificado é o primeiro bit do campo *Flags* (*Reserved Bit*), se este estiver em "0" e o campo *Fragment Offset* estiver preenchido, então um procedimento IPv4 de fragmentação foi realizado. Nesse caso, o *CER_EncDif* utiliza um mecanismo de encapsulamento para adicionar a marca, e o novo destino do pacote é o endereço do *RD* (mesmo *RD* que o endereço IP de destino original do pacote atingiria na área OSPF). Se o *Reserved Bit* estiver em "0" e o campo *Fragment Offset* estiver vazio, então o *CER_EncDif* marca o pacote com *CER_Marca,* define o *Reserved Bit* com "1" e realiza o desvio. Se o *CER_EncDif,* em um roteador ≠ *RO*, verificar que o bit está em "1" e uma falha na rota de desvio existir, isto indica que um processo de desvio já foi realizado, e nesse caso os pacotes são descartados para evitar instabilidades, pois indica a existência de múltiplas falhas independentes (diferentes do tipo SRLG, que são falhas dependentes).

No caso do IPv6, verifica-se se o campo *Flow Label* está vazio. Se estiver, o *CER_EncDif* marca o pacote definindo os três primeiros bits desse campo com "111", conforme [Tang et al. 2002], e os últimos 16 bits recebem a *CER_Marca*. Se o campo *Flow Label* está preenchido com os três primeiros bits com "111" e, em um roteador ≠

RO possuir uma falha na rota de desvio, então um processo de desvio já foi realizado com *CER_EncDif* e nesse caso, descarta-se os pacotes para evitar maiores instabilidades já que existem múltiplas falhas independentes. Caso o campo *Flow Label* esteja marcado com os três primeiros bits diferentes de "111", então houve uso do campo *Flow Label* por parte das máquinas fim a fim e nesse caso, utiliza-se o mecanismo de encapsulamento para adicionar a marca no campo *Flow Label* com os três primeiros bits com "111" e o pacote é então destinado ao *RD*.

É importante salientar que o mecanismo de encapsulamento somente é utilizado quando os campos dos cabeçalhos não puderem ser usados. Acredita-se que esses campos tendem a não ser mais usados pelos tráfegos atuais, pois, no caso do IPv4, o *Fragment Offset* é sempre evitado com o *PMTU* [Mathis and Heffner 2007]. No caso do IPv6, o *DiffServ* usa o campo *Class of Service* [Nickols et al. 1998], que melhor consegue definir parâmetros de QoS em relação ao *Flow Label*.

Se não existir uma falha adjacente, o processo *CER_EncDif* somente é acionado se um pacote marcado com *CER_Marca* entra em um roteador. Nesse caso, o *CER_EncDif* encaminha para a *IR* respectiva ao par *CER_Marca/IR*. Quando a marca do pacote não é encontrada nos pares *CER_Marca/IR* (indica que atingiu o último roteador do *CER_Vetor*, i.e. *RC*), então o encaminhamento padrão baseado apenas no endereço IP de destino (*next-hop*) é reutilizado pelo *CER_EncDif*. A marca no pacote permanece e o encaminhamento baseado no IP de destino é mantido pelo *CER_EncDif* até que atinja *RD*, onde se retira a marca dos pacotes (no caso do IPv4 define também o *Reserved Bit* com "0") ou desencapsula os pacotes. A manutenção da marca serve para o *CER_EncDif* evitar novos desvios em caso de múltiplas falhas independentes.

Em ambiente de múltiplas falhas independentes, o processo *CER_EncDif* evita o surgimento de instabilidades na rede identificando essa ocorrência e agindo com o descarte dos pacotes já marcados. De forma contrária, as abordagens NotVia e a LFA original revelam a possibilidade de um ambiente instável não agindo sobre esse problema.

2.2.3 CER-EXT

No caso dos *CER_Vetores* gerados pelos níveis ECMP e LFA, o *CER_EncDif* desvia os pacotes baseados na marca obtida até *RC=RV*. Entretanto, os *CER_Vetores* gerados pelo nível SIG definem uma seqüência de roteadores até um *RC* após o *RV*, e nesse caso o *CER_EncDif* desses roteadores necessita encaminhar os pacotes baseado na *CER_Marca/IR* para que alcancem o *RC*. Esse desvio é necessário para evitar um comportamento instável gerado pelo encaminhamento padrão IP com rotas OSPF. Para tanto, o *CER_EncDif* necessita de informações para que consiga realizar o desvio com a *CER_Marca*. Essas informações são obtidas pelo processo *CER-EXT*.

O *CER-EXT* é uma extensão do cálculo dos CERs realizado em cada roteador definido na seção 2.1. Para descrever seu funcionamento, há de se considerar um roteador X, que após realizar o cálculo dos CERs, ele deve identificar quais *CER_Vetores* de nível SIG gerados pelos demais roteadores utilizam X como *RV, roteadores_intermediários* ou *RC*. O roteador X então simula cada roteador na base de informações do estado do enlace, diferente de X, como sendo o *RO* e aplica um cálculo dos CERs para obter apenas os *CER_Vetores* que usam X na seqüência de roteadores. Em seguida, X pode identificar qual o par *CER_Marca/IR* a ser usado pelos *CER_Vetores* encontrados

através do *Não_RO_caso* (descrito na seção 2.2.1). A *CER_Marca* é, com isso, a mesma obtida pelos roteadores remotos que geram os *CER_Vetores* no *RO_caso* e também as mesmas no *Não_RO_caso* sem a necessidade de troca de mensagens de rede entre eles.

2.2.4 Manutenção do processo *CER_EncDif*

Quando uma falha é detectada, o processo *CER_EncDif* é acionado no *RO* para marcar e encaminhar os pacotes de acordo com a *IR*. O processo *CER_EncDif* mantém esse encaminhamento no *RO* por um período de tempo suficiente para que todos os processos OSPF dos roteadores consigam atualizar suas FIBs. Este período de tempo pode ser aproximadamente definido utilizando o intervalo gerado pela abordagem *oFIB* [Francois and Bonaventure 2006]. A abordagem *oFIB* define um intervalo de tempo para atualizar a FIB no roteador adjacente à falha (*RO*), sendo que esse intervalo é suficiente para os demais roteadores da topologia afetados indiretamente por essa falha (árvore SPF afetada) completem a atualização de suas FIBs. Somente após esse intervalo o *RO* atualiza a FIB e então o processo *CER_EncDif* interrompe automaticamente a marcação e desvio dos pacotes no *RO*, pois a entrada na FIB contém agora um *next-hop* correto não acionando o *CER_EncDif* para desviar os pacotes.

Somente quando o *CER_EncDif* interrompe sua marcação no *RO* que um novo cálculo dos CERs é realizado para obter *CER_Vetores* de acordo com o novo estado da topologia. Os demais roteadores da topologia, já com as rotas atualizadas na FIB, também realizam um novo cálculo dos CERs apenas quando não existir mais o fluxo de pacotes marcados e desviados em *RO*. Os pacotes marcados são encaminhados com a marca (ao atingir *RD* a marca é retirada), e se não existir a marca, então são corretamente encaminhados baseados no IP de destino (agora com as rotas atualizadas na FIB).

Durante o processo de desvio, se os pacotes marcados afetarem outros fluxos não afetados pela falha (possibilidade de congestionamentos), o *CER_EncDif* impede essa ocorrência com a seguinte regra: somente encaminhar os pacotes marcados se o tamanho da fila for menor que 80%, caso contrário, deve-se descartar esses pacotes. Em todos os testes simulados, o valor definido de 80% foi provado ser suficiente para impedir o comprometimento dos demais fluxos de pacotes quando o *CER_EncDif* estava atuando.

3. Avaliação do E-CER

Um algoritmo para E-CER foi implementado para gerar os *CER_Vetores* por cada roteador de uma topologia obedecendo à abordagem descrita na seção 2. A complexidade do algoritmo é $O(n^3 log(n))$, já incluindo o processo *CER-EXT*, onde n é o número de roteadores na área OSPF. Devido às restrições de espaço, maiores detalhes do algoritmo não são revelados. Esse algoritmo foi adaptado no simulador JavaSim (http://www.j-sim.org). A escolha desse simulador deve-se ao código OSPF portado do *GNU Zebra Project* (http://www.zebra.org), que demonstrou ser muito próximo do comportamento real do OSPF. Como o esquema E-CER utiliza as informações do estado do enlace já atualizadas pelo OSPF, o cálculo dos CERs é realizado em *background* e não influencia o processo de encaminhamento das rotas OSPF. Com os resultados desse cálculo disponíveis na FIB (*CER_Marca/IR*), se ocorrer uma falha, apenas o processo *CER_EncDif* é executado. Para realizar a avaliação, somente a abordagem NotVia foi implementada seguindo a descrição de [Bryant et al. 2007], pois essa é a

única abordagem que atinge 100% de cobertura de falha de enlace, roteador, ou SRLG.

Ao contrário do E-CER, o NotVia apenas recomenda o uso das abordagens originais de ECMP ou LFA [Bryant et al. 2007], não adaptando as mesmas para melhor utilizá-las (por exemplo, a abordagem LFA não identifica qual o melhor caminho de recuperação se existirem vários possíveis, já a abordagem E-CER, nesse caso, seleciona o caminho que conter o menor número de roteadores).

Foram geradas 50 topologias artificiais com o gerador BRITE (http://www.cs.bu.edu/brite) obedecendo aos mesmos parâmetros usados em [Hansen et al. 2006] e 8 topologias reais de *backbone* (Abilene, AGIS, ATHome, AT&T, CAIS, GEANT2, Qwest e Servint) para avaliar o E-CER em relação ao NotVia. A primeira avaliação baseia-se em uma avaliação conduzida por [Hansen et al. 2006], que confronta a extensão dos caminhos de recuperação em termos do número de roteadores percorridos entre o roteador origem até o roteador destino. Esta comparação permite identificar qual a extensão seguida pelos pacotes desviados.

A Figura 2 mostra os resultados dessa primeira avaliação para ambas as abordagens. As 50 topologias artificiais geradas com BRITE geraram resultados similares e são sintetizados em uma figura apenas. Em todas as topologias, E-CER resulta em caminhos freqüentemente menores que NotVia, pois os pacotes desviados com NotVia seguem um caminho encapsulado até *next-next-hop*, e somente a partir desse roteador, os pacotes seguem o caminho correto até o roteador destino. Esse fato tem uma maior ocorrência nas topologias AT Home, AGIS e CAIS, pois possuem seqüências de roteadores em série, o que força um caminho maior para desviar dos pacotes até o *next-next-hop* com NotVia. O E-CER, no pior caso, pode gerar o mesmo caminho que o NotVia e esse fato pode ocorrer quando existe uma alta conectividade entre os roteadores, no entanto, essa ocorrência é dependente de topologia e não foi apresentada nas topologias testadas.

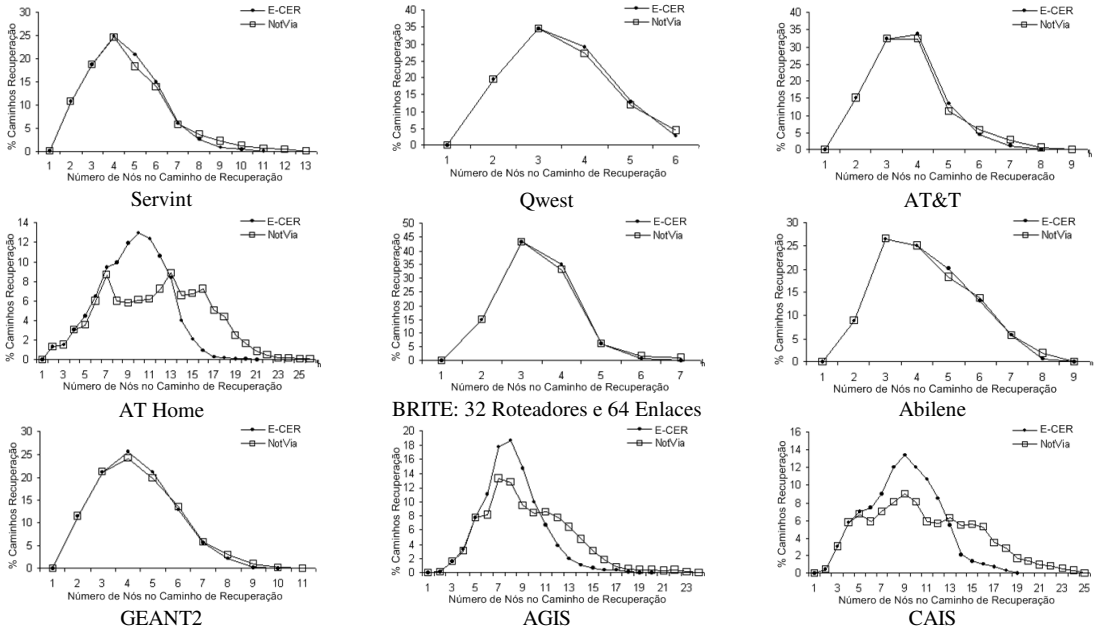


Figura 2. Número de Roteadores nos Caminhos de Recuperação por Topologia

Como a extensão do número de roteadores nos *CER_Vetores* e a geração dos

pares *CER_Marca/IR* são minimizados na seção 2, a quantidade de informações adicionadas na FIB é conseqüentemente reduzida. A segunda avaliação é a quantidade de informações adicionadas na FIB pelo E-CER e pelo NotVia. A Tabela 1 revela uma estimativa da quantidade de informações extras adicionadas na FIB por roteador, pois não foi possível obter os dados reais sobre a quantidade de entradas na FIB por roteador das topologias usadas. Usa-se nessa Tabela 1 o seguinte padrão: o número de entradas OSPF na FIB (nEO), *CER_Marca/IR* (CIR) com 3 bytes (8 bytes + 1 byte), *Ref* (seção 2.2.1) com 1 byte, cada nova entrada na FIB (nFIB) necessita de 12 bytes [Moy 1998], e um *next-next-hop* (*n-n-h*) é um endereço IP com 4 bytes [Bryant et al. 2007]. A abordagem E-CER gera uma quantidade de informações reduzida na FIB comparada com o NotVia, pois o E-CER necessita do *CER_Marca/IR* referenciado entre as entradas OSPF na FIB através do campo *Ref*, enquanto que o NotVia necessita de novas entradas na FIB para comportar o *n-n-h* além da adição desses *n-n-h* para cada entrada OSPF na FIB.

Tabela 1. Quantidade de Informação Extra adicionada na FIB

	E-CER	NotVia
Servint	12(CIR) + nEO(Ref), Total: 36 + nEO bytes	58(nFIB) + nEO(n-n-h) , Total: 696 + 4(nEO) bytes
Qwest	10(CIR) + nEO(Ref), Total: 30 + nEO bytes	93(nFIB) + nEO(n-n-h) , Total: 1116 + 4(nEO) bytes
AT&T	11(CIR) + nEO(Ref), Total: 33 + nEO bytes	72(nFIB) + nEO(n-n-h), Total: 864 + 4(nEO) bytes
AT Home	36(CIR) + nEO(Ref), Total: 108 + nEO bytes	106(nFIB) + nEO(n-n-h), Total: 1272 + 4(nEO) bytes
BRITE	10(CIR) + nEO(Ref), Total: 30 + nEO bytes	128(nFIB) +nEO(n-n-h) , Total: 1536 + 4(nEO) bytes
Abilene	9(CIR) + nEO(Ref), Total: 27 + nEO bytes	30(nFIB) + nEO(n-n-h) , Total: 360 + 4(nEO) bytes
GEANT2	12(CIR)+ nEO(Ref), Total: 36 + nEO bytes	76(nFIB) + nEO(n-n-h) , Total: 912 + 4(nEO) bytes
AGIS	31(CIR) + nEO(Ref), Total: 93 + nEO bytes	149(nFIB) +nEO(n-n-h) , Total: 1788 + 4(nEO) bytes
CAIS	30(CIR) + nEO(Ref), Total: 90 + nEO bytes	88(nFIB) +nEO(n-n-h) , Total: 1056 + 4(nEO) bytes

A terceira avaliação verifica qual a taxa de redução de pacotes perdidos durante o período de convergência do OSPF quando com o uso da abordagem E-CER. Algumas mudanças foram necessárias no JavaSim para modificar o código OSPF de forma a ter um período de convergência aproximado de 200ms além de retardar o processo de sinalização de falha em 20ms, conforme [Francois et al. 2005] (antes de 20ms, os pacotes são descartados). Utilizou-se apenas a representação da topologia GEANT2 (ilustrado na Figura 3) para essa análise, pois se conhecia as capacidades reais dos seus enlaces (gigabit), de forma que os seus pesos são valores inversamente proporcionais às suas capacidades e o período de convergência do OSPF pode alcançar 200 ms [Francois et al. 2005].

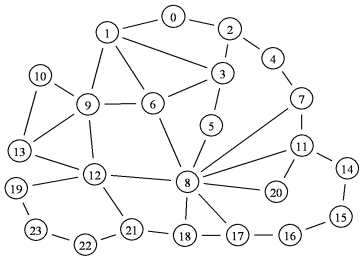


Figura 3. GEANT2

Planejou-se uma matriz de tráfego de forma que todos os enlaces da topologia ocupassem no máximo 50% de suas capacidades (ver seção 2). Entre os fluxos de tráfego, a avaliação ocorre em 6 pares de fluxos (origem, destino): (2,18), (18,2), (1,17), (17,1), (9,11) e (11,9), pois estes usam dois dos roteadores centrais da topologia (6 e 8)

sendo o pior ambiente de falha. Esses 6 fluxos são gerados a uma taxa de bits constante de 160Mbps e pacotes com tamanho 256 bytes para ajustar o enlace 6-8 a 480 Mbps.

Definiram-se 3 cenários de falha: enlace 6-8, roteador 6, e roteador 8. Primeiramente a avaliação ocorre com o OSPF (convergência ≈ 200ms) e depois com E-CER auxiliando o OSPF durante esse período de convergência. Todos os fluxos de pacotes analisados são desviados e conseguem se ajustar à largura de banda remanescente (50% livre). Dessa forma, é possível analisar isoladamente a taxa de pacotes perdidos gerado exclusivamente durante o período de convergência, o que não pode ser medido corretamente em um ambiente instável com uma alta taxa de tráfego na topologia. Entretanto, o *CER_EncDif* consegue impedir um ambiente mais instável (com congestionamentos) descartando os pacotes marcados no momento que a capacidade das filas for > 80% (seção 2.2.4). Uma regra similar não existe em qualquer outra abordagem IPFRR, porém é uma medida recomendada [Shand and Bryant 2007].

A Tabela 2 revela a porcentagem de pacotes perdidos durante o período de convergência. O OSPF tem a maior taxa de pacotes perdidos devido à sua característica reativa, mesmo com um período de convergência de ≈ 200ms. A abordagem E-CER auxilia o OSPF durante esse período, pois durante ≈ 200ms, o processo *CER_EncDif* realiza o encaminhamento para desviar os pacotes baseados no par *CER_Marca/IR*, o que explica a baixa taxa de pacotes perdidos. Após 200ms, os processos OSPF dos roteadores adjacentes à falha são os últimos a atualizar as rotas na FIB (utilizando *oFIB*, ver seção 2.2.4) e o *CER_EncDif* não é mais acionado.

Tabela 2. % Pacotes Perdidos Durante o Período de Convergência

	OSPF (≈ 200ms)			E-CER + OSPF (≈ 200ms)		
	Enlace 6-8	Roteador 6	Roteador 8	Enlace 6-8	Roteador 6	Roteador 8
Fluxo (2,18)	23,4 %	24,5 %	25,4 %	1,8 %	2,8 %	2,7 %
Fluxo (18,2)	23,2 %	24,1 %	25 %	1,4 %	2,6 %	2,6 %
Fluxo (1,17)	23,5 %	24,7 %	24,7 %	1,6 %	2,7 %	2,5 %
Fluxo (17,1)	23,1 %	24,7 %	25,1 %	1,5 %	2,6 %	2,6 %
Fluxo (9,11)	23,1 %	24,3 %	25,3 %	2,1 %	2,5 %	2,7 %
Fluxo (11,9)	23,2 %	24,2 %	25,4 %	1,3 %	2,6 %	2,6 %

4. Conclusão

Dependendo da aplicação utilizada, os protocolos de roteamento não reagem à uma falha em tempo hábil, mesmo reduzindo o período de convergência em menos de 1 segundo. Isto pode acarretar em uma alta taxa de pacotes perdidos comprometendo a confiabilidade da rede. Propõe-se a nova abordagem E-CER para gerar previamente caminhos de recuperação como auxílio ao OSPF. O E-CER toma como base o *framework* IPFRR, porém realizando adaptações para minimizar os caminhos e a quantidade de roteadores necessários para desvio dos pacotes. Essa característica resulta em caminhos de recuperação menores, com reduzida representação na FIB e que se mostrou capaz de reduzir a taxa de pacotes perdidos durante o período de convergência. Dependendo da falha, cada caminho obtido com o E-CER é sempre o mesmo caminho obtido com OSPF após o período de convergência, o que permite uma adaptação gradual dos roteadores às novas rotas do OSPF. Uma implementação em ambiente real, bem como aspectos de segurança e uma versão para tratar múltiplas falhas independentes serão abordados em trabalhos futuros.

References

Ahuja, R. K., Magnanti, T. L., Orlin, B. O. (1993) Network Flows: Theory, Algorithms, and Applications, Prentice-Hall.

Atlas, A., and Zinin, A. (2007). "Basic Specification for IP Fast-Reroute Loop-Free Alternate". In *Internet Draft*. IETF Routing-WG.

Barreto, F., Wille, E. C. G., and Nacamura Jr, L. (2007) "Contornando Falhas em Backbones IP com Caminhos Emergenciais Rápidos", In *SBRC 2007 Workshop de Teste e Tolerância a Falhas*.

Basu, A. and Riecke, J. F. (2001). "Stability Issues in OSPF Routing". In *SIGGCOM Applications, technologies, arch., and protocols for communications*, pages 225-236.

Bryant, S., FilsFils, S., Previdi, S., and Shand, M. (2005). "IP Fast Reroute Using Tunnels". In *Internet Draft*. IETF Routing-WG.

Bryant, S., Shand, M., and Previdi, S. (2007). "IP Fast Reroute Using Not-via Address". In *Internet Draft*. IETF Routing-WG.

Francois, P., and Bonaventure, O. (2006). "Avoiding Transient Loops during IGP Convergence in IP Networks". In *IEEE INFOCOM Computer Communications*, pp. 237-247

Francois, P., Filfis, C., Evans, C., and Bonaventure, O. (2005). "Achieving sub-second IGP convergence in large IP networks". In *ACM SIGCOMM*, pages 34-44.

Hansen, A. F., and Cicic, T., and Gjessing, S. (2006). "Alternative Schemes for Proactive IP Recovery". In *Next Generation Internet Desing and Engineering,* pages 1-8.

Iannaccone, G., Chuah, C., Bhattacharyya, S., and Diot, C. (2004). "Feasibility of IP Restoration in a Tier-1 Backbone". In *IEEE Network Magazine*, pages 13-19.

Kvalbein, A., Hansen, A. F., Cicic, T., Gjessing, S., and Lysne, O. (2006). "Fast IP Network Recovery using Multiple Routing Configurations". In *IEEE INFOCOM Computer Communications*, pages 1-11.

Mathis, M., and Heffner, J. (2007). "Packetization Layer Path MTU Discovery". In *RFC 4821*, IETF Network-WG.

Moy, J. (1998). "OSPF version 2". In *RFC 2328*. IETF Network-WG.

Narvaez, P. (2000). Routing Reconfiguration in IP Networks. Phd Thesis. Massachusetts Institute of Technology.

Nelakuditi, S., Lee, S., Yu, Y., and Zang, Z. (2007) "Fast Local Rerouting for Handling Transient Link Failures". In *IEEE Transactions on Networking*, pages 359-372.

Nickols, K., Blake, S., Baker, F. Black, D. (1998) "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers". In *RFC 2474*. IETF Network-WG.

Shand, M., and Bryant, S. (2007). "IP Fast Reroute Framework". In *Internet Draft*. IETF Routing-WG.

Tang, X., Tang, J., Huang, G., and Siew, C. (2002). "QoS Provisioning Using IPv6 Flow Label in the Internet". In *IEEE Conference on Communication and Signal Processing*.

Protocolo de Transporte Colaborativo para Redes de Sensores sem Fio

Eugênia Giancoli[1,2], **Filippe C. Jabour**[1,2], **Aloysio de Castro Pinto Pedroza**[1]

[1]Programa de Engenharia Elétrica – Universidade Federal do Rio de Janeiro (UFRJ)
Rio de Janeiro – RJ – Brasil

[2]Centro Federal de Educação Tecnológica de Minas Gerais (CEFET-MG)
Leopoldina – MG – Brasil

Abstract. *This work presents the Collaborative Transport Control Protocol (CTCP), a new transport protocol for sensor networks. It provides end-to-end reliability and adapts to different type of applications through its mechanism of reliability variation. Its congestion control and detection differentiates communication losses of buffer overflow. It's called collaborative since all nodes detect and act on congestion control and its distributed storage responsibility. It is scalable and independent of the underlying network layer. It was observed that the distributed fault recovery increases the reliability and that the duplication of storage responsibility minimizes the possibility of definitive loss of messages.*

Resumo. *Este trabalho apresenta o CTCP, um novo protocolo de transporte para redes de sensores. Ele provê confiabilidade fim-a-fim e se adapta aos diferentes tipos de aplicação através do mecanismo de variação desta confiabilidade. Seu controle e detecção de congestionamento diferencia as perdas relativas a erro de transmissão daquelas relativas ao esgotamento de buffers. É chamado colaborativo já que todos os nós detectam e atuam sobre o congestionamento e em função da responsabilidade distribuída de armazenamento. É escalável e independe das camadas de rede subjacentes. Observou-se que a recuperação distribuída de falhas aumenta a confiabilidade e que a duplicação desta responsabilidade minimiza a possibilidade de perda definitiva de mensagens.*

1. Introdução

As redes de sensores sem fio (RSSF) fornecem uma solução de sensoriamento amplamente distribuída e econômica para ambientes onde as redes tradicionais não conseguem atuar. Suas principais aplicações estão em monitoramento militar, ambiental, médica, industrial ou infraestruturas domésticas.

Atualmente, o hardware das redes de sensores é escolhido visando maximizar a vida útil da rede para uma aplicação específica. Contudo, acreditamos que o número de aplicações para redes de sensores crescerá e existirá a necessidade de hardwares mais poderosos e genéricos que poderão ser reprogramados durante sua vida útil. Esta reprogramação, é uma das aplicações que requer um protocolo de transporte que garanta confiabilidade na entrega dos dados.

Portanto, cada aplicação possui diferentes características e requisitos de tipos de dados, taxas de transmissão e confiabilidade. A maioria dos protocolos existentes para a camada de transporte das redes de sensores foram adaptados para funcionar com determinados tipos de aplicações, ou assumem que os nós trabalham com determinadas camadas de rede ou enlace. Como resultado, estes protocolos não podem ser aplicados em qualquer tipo de rede de sensores.

Assim, é necessário projetar um protocolo da camada de transporte que possa suportar aplicações múltiplas na mesma rede, provendo controle de confiabilidade variável, controle de congestionamento, redução de perdas e suporte a desconexões freqüentes [Kim et al. 2007]. Por isso, este trabalho explora as decisões de projeto de um protocolo de transporte para suportar uma classe de aplicações que requer entrega de dados confiável nas redes de sensores sem fio.

De acordo com [Iyer et al. 2005], os requisitos básicos para uma camada de transporte genérica para redes de sensores são:

Heterogeneidade: os nós sensores podem possuir múltiplos sensores (luz, temperatura, presença, etc) com características diferentes de transmissão. Os pacotes gerados por cada sensor para uma aplicação constitui seu fluxo de dados, que pode ser contínuo ou baseado em eventos. Nas aplicações de fluxos contínuos, os nós transmitem pacotes periodicamente para a estação base. Nas aplicações baseadas em evento, os nós transmitirão dados somente quando um determinado evento ocorrer. Os dois tipos de fluxo podem existir na mesma rede e o protocolo da camada de transporte deve suportar múltiplas aplicações heterogêneas dentro da mesma rede.

Confiabilidade: as aplicações têm requisitos diferentes de confiabilidade. Por exemplo, em ambiente militar, os dados transmitidos pelos sensores devem chegar sempre à estação base. Na reprogramação de grupo de sensores, os dados também precisam chegar até todos os nós. Entretanto, no monitoramento de temperatura, alguns pacotes podem ser perdidos sem causar grandes danos. O protocolo de transporte deve explorar esta diferença visando economizar energia.

Controle de congestionamento: todos os nós da rede geram pacotes que convergem para os nós localizados ao redor da estação base. Este nós encaminham mais pacotes e consequentemente, aumenta a possibilidade de congestionamento perto da estação base. Altas taxas de dados, rajadas de dados e colisões são as outras razões para os congestionamentos nas redes de sensores.

O protocolo apresentado por este trabalho, chamado Collaborative Transport Control Protocol (CTCP), fornece um transporte confiável, escalável e genérico às RSSF onde cada nó pode ser a origem de vários fluxos com diferentes características. Além disso, suporta redes com aplicações múltiplas e provê funcionalidades adicionais como detecção e controle de congestionamento e variação da confiabilidade.

Na seção 2 serão citados os trabalhos relacionados, na seção 3 o CTCP será descrito, especificado e modelado probabilisticamente. Na seção 4 finalizamos com as conclusões e trabalhos futuros.

2. Trabalhos relacionados

O primeiro protocolo analisado foi o TCP[Allman et al. 1999]. Este protocolo faz parte da pilha de protocolos TCP/IP que foi teoricamente projetada para operar de forma independente das tecnologias das camadas inferiores. Assim, o perfil de protocolos TCP/IP deve operar em redes cabeadas confiáveis, redes sem fio, redes de satélite, redes ópticas etc. No entanto, os atuais mecanismos do TCP se baseiam em suposições típicas de redes cabeadas convencionais, tais como a existência de uma conectividade fim-a-fim entre fonte e destino durante todo o período correspondente à sessão de comunicação, atrasos de comunicação relativamente pequenos (na ordem de milissegundos), baixas taxas de erros, mecanismos de retransmissão efetivos para reparar erros e suporte a taxas de dados bidirecionais relativamente simétricas.

Desta forma, o TCP não se adequa às redes de sensores. Estas redes são caracterizadas por atrasos longos ou variáveis, quebra freqüente de conexões, conectividade intermitente, altas taxas de erro e limitação de recursos. A pilha TCP/IP apresenta um baixo desempenho nestas redes.

Reliable Multi-Segment Transport (RMST) [Stann and Heidemann 2003] foi projetado para funcionar em conjunto com o Direct Diffusion Protocol [C. Intanagonwiwat 2000], ou seja, existe uma dependência da camada de rede. O RMST é um protocolo baseado em NACK e trabalha com ou sem cache. Quando o cache está habilitado, os nós intermediários armazenam os fragmentos de dados, o que pode causar esgotamentos dos *buffers*. RMST não garante a confiabilidade quando um nó falha depois de receber e antes de transmitir os fragmentos de dados. Além disso, o RMST não trata o congestionamento de dados na rede de sensores.

Event-to-Sink Reliable Transport (ESRT)[Sankarasubramaniam et al. 2003] foi projetado para redes centradas em aplicações. É pressuposto que a estação base está interessada em um determinado evento que pode ser detectado por vários nós ao mesmo tempo. No ESRT, é a estação base que faz o controle do congestionamento, socilitanto aos nós o aumento ou diminuição da taxa de transmissão. Não garante entrega fim-a-fim. Em redes reais os nós só transmitem dados quando detectam um evento, o que dificulta o controle da taxa de transmissão pela estação base.

Pump Slowly, Fetch Quick (PSFQ)[Wan et al. 2005] tem como objetivo reprogramar os nós de uma rede de sensores. Possibilita um broadcast de dados confiável da estação base para os nós. Contudo, possui alto consumo de energia, uma vez que a confiabilidade é alcançada através do aumento de retransmissões na rede.

Sensor Transmission Control Protocol (STCP)[Iyer et al. 2005] foi projetado para ser um protocolo genérico. Entende-se por genérico a capacidade de se adaptar a qualquer tipo de aplicação e trabalhar com qualquer camada de rede subjacente. Possui controle de congestionamento, uma vez que os nós ao redor da estação base podem estar sujeitos a esgotamento de *buffers*. Possui nível de confiabilidade adaptável visando economia de energia. Contudo, quase todos os seus controles baseiam-se fortemente na estação base, que possui amplos poderes computacionais e energéticos. Tal comportamento se desvia do desafio maior das redes de sensores que é o auto-gerenciamento. Considera-se questionável, ainda, que o nível de confiabilidade requerido pela aplicação seja controlado pelo nó que origina a informação, pois, o conhecimento global da rede e da aplicação seria

necessário para tomar tal decisão. O sincronismo de tempo da rede de sensor é requerido para economizar energia em aplicações que possuam fluxos de dados contínuos. Contudo, não existem resultados que provem que o overhead causado pela sincronização justifique esta escolha.

Neste trabalho apresentamos o CTCP. Trata-se de um protocolo colaborativo de transporte baseado em mecanismos conhecidos [Allman et al. 1999] de reconhecimento de pacotes (ACK) e temporização. Nosso trabalho utiliza o reconhecimento salto-a-salto, demonstrado eficiente em [Stann and Heidemann 2003] mas prevê a liberação imediata do cache (*buffers*) do nó, aumentando a sua capacidade de reencaminhamento e evitando o congestionamento. O CTCP dispensa ainda a sincronização da rede, utilizada em [Iyer et al. 2005] e é capaz de suportar quebra das conexões sem perda de dados. Mesmo quando um nó recebe os dados e falha antes de reencaminhá-los, o protocolo é capaz de recuperar-se desta perda. Foi projetado para trabalhar com qualquer tipo de camada subjacente e possui um controle de congestionamento capaz de evitar perdas relativas a esgotamento de *buffers*. Os dois níveis de confiabilidade garantem ao CTCP flexibilidade para se adaptar aos diferentes tipos de aplicações.

3. Especificação do protocolo

As principais contribuições do CTCP são:

- Garantir que todos os segmentos sejam entregues, à camada de aplicação da estação base, mesmo na presença de falhas de nós e freqüentes desconexões
- Adaptação ao perfil de confiabilidade da aplicação visando economizar energia
- Diferenciar a perda relativa a congestionamento da perda relativa a erro de transmissão
- Controlar o congestionamento através da interrupção/liberação imediata do encaminhamento
- Independência das camadas subjacentes

Para a modelagem e análise formal deste protocolo, foram empregados dois métodos, um formalismo matemático denominado Redes de Petri Predicado Ação [Nielsen et al. 1981] e uma linguagem de especificação chamada CCS (*A Calculus of Communicating Systems*), de Robin Milner [Milner 1980]. São métodos de especificação formal que permitem o desenvolvimento de sistemas com um mínimo de ambigüidades, através de uma sintaxe e semântica bem definidas. A especificação formal do nosso protocolo possibilita uma análise de comportamento funcional de certas propriedades, como ausência de bloqueios(*deadlocks*), sincronismo e seqüência correta de mensagem, além da verificação da consistência da especificação.

3.1. Abertura e encerramento da conexão fim-a-fim

Antes de iniciar a transmissão de dados, um pacote de abertura de conexão (ABR) é enviado da origem para a estação base. Este pacote informa à estação base o identificador do fluxo de dados e o primeiro número de seqüência. Quando a estação base recebe este pacote, ela reserva os *buffers*, inicializa as variáveis necessárias e envia um reconhecimento (ACK) à origem da conexão. No cabeçalho do ACK são especificados o nível de confiabilidade requerido pela aplicação à qual o fluxo pertence e o identificador da conexão (ID), que é controlado pela estação base para evitar que existam conexões diferentes com

o mesmo ID. Controlar o identificador da conexão provê flexibilidade ao protocolo, uma vez que, este dado será necessário para implementar a multiplexação de diferentes fluxos da rede. Esta implementação será tratada em trabalhos futuros.

Logo a seguir, o nó origem estará apto a iniciar a transmissão de dados para a estação base. Para que este protocolo fosse o mais genérico possível, durante seu projeto, preocupou-se em estabelecer formatos de pacotes compatíveis com a maioria das redes subjacentes. Desta forma, foi preciso considerar que a comunicação sem fio, e principalmente as redes de sensores, possuem dificuldades de transferência de pacotes que sejam maiores do que o quadro da camada de enlace. Apesar de alguns protocolos, como o 802.11 [Vassis et al. 2005], possuirem fragmentação e agrupamento, existem limites no tamanho dos pacotes que uma entidade consegue fragmentar e garantir a entrega. [Stann and Heidemann 2003]. Por isso, os sensores que utilizarão o CTCP serão pré-configurados com o MSS (Maximum Segment Size) permitido pelas camadas de rede subjacente. Tal variável não precisa ser negociada durante a abertura da conexão.

Neste protocolo, ao contrário do STCP, a confiabilidade não é definida pelo nó origem, uma vez que o próprio nó não possui ”inteligência“ suficiente para identificar o tipo de confiabilidade requerida por cada aplicação.

Esta conexão fim-a-fim não expira por tempo nem por ausência de tráfego podendo atender a requisitos de desconexões por longos períodos de tempo sem perda de dados. A figura 1 ilustra a abertura e o fechamento das conexões.

Quando a camada de aplicação do nó origem termina seu trabalho envia um pacote de fechamento de conexão à estação base. A estação base, então, libera os *buffers* e variáveis da conexão especificada e responde com um ACK. A origem aguarda o recebimento do ACK para efetuar a interrupção da conexão. Resumidamente, o fechamento da conexão consiste em informar à estação base que a origem terminou de gerar dados e não vai transmitir mais nenhum pacote, ou se o desejar fazer abrirá uma nova conexão.

3.1.1. Abertura e Encerramento de conexões no CTCP descritas em CCS

$$Abertura\_Encerramento = Origem || Base$$

$$Origem = !abr.Esp\_ack$$
$$Esp\_ack = ?ack.T\_dados\_or$$
$$T\_Dados\_or = !disc.Esp\_ack\_final$$
$$Esp\_ack\_final = ?ack.Origem$$
$$Base = ?abr.!ack.T\_Dados\_Base$$
$$T\_Dados\_Base = ?disc.!ack.Base$$

3.2. Controle da variação da confiabilidade

Cada aplicação possui requisitos diferentes de confiabilidade. Algumas, por exemplo, suportam perdas de dados enquanto outras precisam de banda passante mínima. Desta forma, para especificar a confiabilidade requerida é preciso que se tenha conhecimento da aplicação e seus objetivos. Ao contrário do que sugere o trabalho [Iyer et al. 2005], nesta proposta, esta decisão não será tomada pelos nós, uma vez que eles não possuem uma visão global da aplicação.

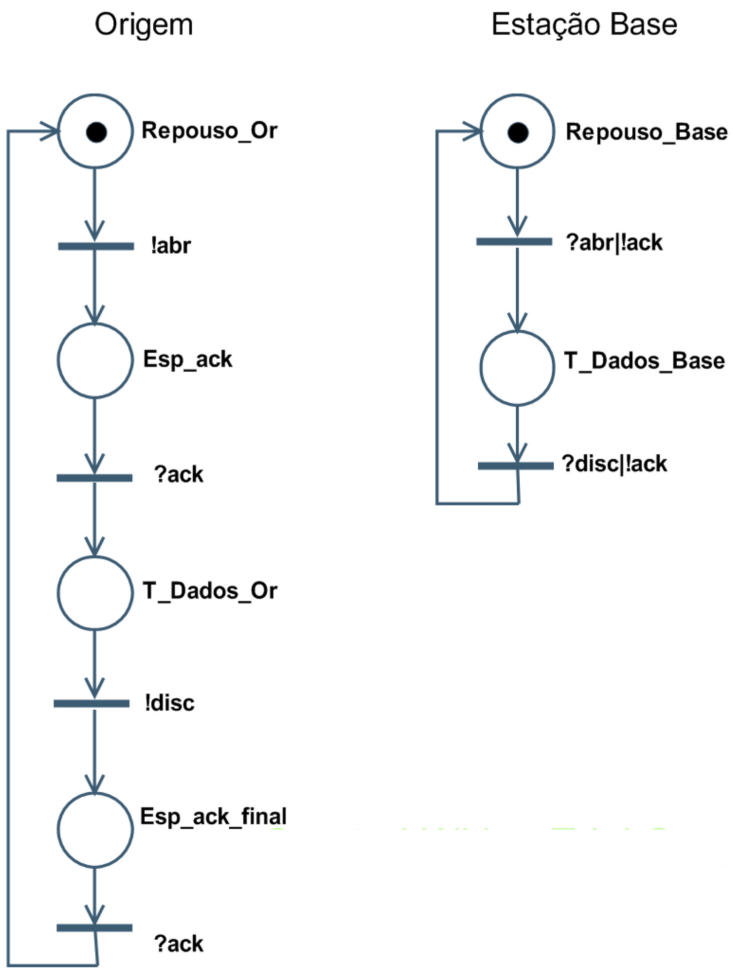


Figura 1. Abertura e Fechamento das conexões

Neste trabalho, a estação base, é responsável por estipular o nível de confiabilidade requerido pela aplicação. Além disso, o nível de confiabilidade, que é definido na abertura da conexão, pode ser alterado dinamicamente sempre que houver necessidade. Esta necessidade pode ser representada, por exemplo, pelo esgotamento de energia dos nós. Nessa situação, pode ser mais interessante trabalhar com menos confiabilidade para maximizar a vida útil da rede.

A confiabilidade será medida pela fração de pacotes recebidos pela estação base com sucesso.

É preciso ressaltar que cada pacote pode, ou não, tomar diferentes caminhos para chegar até o destino. Isto depende do algoritmo de roteamento e é tarefa da camada de rede. Este trabalho parte do pressuposto que a rota já foi definida pela camada de rede e que há reconfiguração de rotas no nível de rede em caso de falha ou desligamento dos nós.

Uma vez definido o nível de confiabilidade requerido pela aplicação, os nós da rede poderão agir de duas maneiras diferentes descritas nas seções 3.2.1 e 3.2.3.

3.2.1. Nível 1 de confiabilidade

Este nível de confiabilidade visa economia de energia, através da redução de transmissões, possui baixo custo de *buffers* e aplica-se principalmente a aplicações que possuem alguma redundância de dados ou que possam tolerar perdas.

Depois de receber um pacote de um nó A, o nó B envia ao nó A um reconhecimento (ACK) e passa a ser temporariamente responsável pela entrega do pacote à estação base. Este processo acontecerá repetidamente, através da rota estipulada pela camada de rede, até que a estação base receba o pacote de dados, e envie um ACK ao nó imediatamente anterior. Qualquer um dos nós, ao receber um ACK poderá descartar o pacote enviado, poupando espaço em seus *buffers* conhecidamente reduzidos. Esta situação está representada graficamente na figura 2 e formalmente, através das Redes de Petri na figura 3

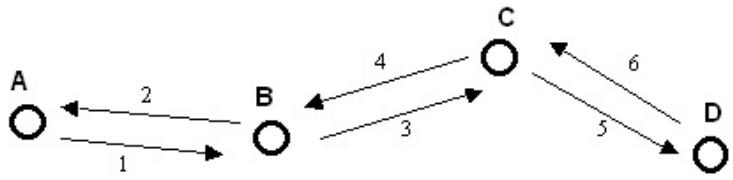


Figura 2. Ordem da troca de mensagens do CTCP para nível 1 de confiabilidade

Repare que, ao assumir a obrigação de entregar o pacote, o nó intermediário deverá manter uma cópia deste pacote em buffer até receber o ACK. A ausência de recebimento do ACK gera um esgotamento de temporização do nó origem e a retransmissão do pacote não reconhecido.

Considere, entretanto, a seguinte situação: o nó A, origem, envia dados ao nó B. Este, por sua vez, recebe os dados, armazena e envia o ACK ao nó A. Neste exato momento o nó B falha e deixa de fazer parte da rede. Logo, os dados que estavam sob responsabilidade do nó B não serão entregues à estação base e o nó A não perceberá esta falha até que a conexão seja fechada. Esta situação pode ser resolvida através do aumento do nível de confiabilidade.

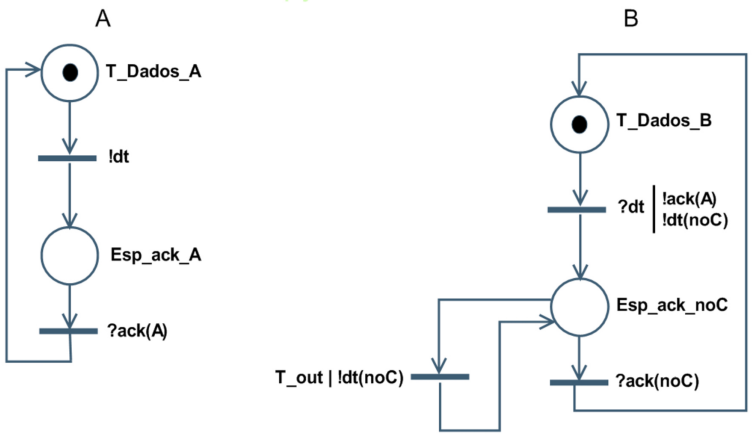


Figura 3. Comunicação entre dois nós da rede

Para escolher entre os dois níveis de confiabilidade, o usuário que interage com a estação base, deverá considerar as necessidades da aplicação.

3.2.2. Confiabilidade Nível 1 descrita em CCS

$Nivel\_1 = A|||B|||Timer$

$A = !dt.Esp\_ack\_A$
$Esp\_ack\_A = ?ack(A).A$
$B = ?dt.!ack.!dt(noC).!starttimer.Esp\_ack\_noC$
$Esp\_ack\_noC = ?ack(noC).!reset\_timer.B + ?t\_out.!dt(noC).!start\_timer.Esp\_ack\_noC$
$Timer = ?start\_timer.(!t\_out.Timer + ?reset\_timer.Timer)$

3.2.3. Nível 2 de Confiabilidade

O nó A envia os dados para o nó B e espera receber o duplo ACK. O duplo ACK é gerado da seguinte maneira: B recebe os dados de A e devolve para A o primeiro ACK. O nó B envia os dados para C que devolve para B o primeiro ACK. Quando B receber o primeiro ACK de C enviará para A o segundo ACK. Somente neste momento A descartará os dados mantidos em buffer. Todos os nós repetirão este processo, sucessivamente, até que os dados cheguem à estação base. A figura 4 representa graficamente esta troca de mensagens.

Se o nó B falhar antes de entregar os dados ao nó C, o nó A não receberá o segundo ACK e retransmitirá o pacote. Observe que partimos do pressuposto de que as falhas dos nós são monitoradas pelo algoritmo de roteamento e este será responsável por refazer a rota quando da falha de um determinado nó, ou conjunto deles.

O protocolo descrito acima possibilita que a mensagem chegue ao seu destino com uma probabilidade maior, uma vez que a falha de um nó no caminho não interrompe a entrega dos dados. A especificação formal do nível 2 é feita de forma análoga à do nível 1.

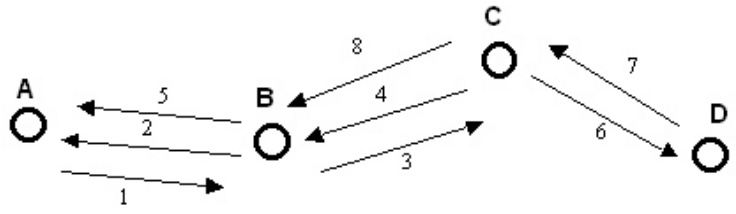


Figura 4. Ordem da troca de mensagens do CTCP para nível 2 de confiabilidade

3.3. Avaliação probabilística da confiabilidade

As redes tradicionais garantem confiabilidade na camada de transporte através de mecanismos de recuperação de erros fim-a-fim. Esta abordagem não é ideal para as redes de sensores pois ao contrário das redes tradicionais onde os nós intermediários são apenas

roteadores, nas redes de sensores eles possuem a camada de transporte o que nos permite distribuir a tarefa de recuperação de erros. Esta colaboração entre os nós torna-se factível porque todos os nós pertencem à mesma entidade administrativa e desejam atingir um mesmo objetivo.

Além de todas as diferenças no modelo de comunicação e serviços das redes de sensores, o maior problema com a recuperação de erros fim-a-fim é a baixa qualidade dos enlaces, pois, os sensores, normalmente trabalham com rádios de baixo alcance, em ambientes com muitos obstáculos e usam técnicas de encaminhamento de mensagens de múltiplos saltos. Desta forma, os erros se acumulam exponencialmente através dos múltiplos saltos.

3.3.1. Probabilidade de entrega fim-a-fim de uma mensagem

Considere que as chances de trocar uma mensagem com sucesso através de um único salto seja p. Assim, a taxa de erros no canal de comunicação é dada por $(1 - p)$. Seja q a probabilidade de sucesso no envio de um ACK por um único salto. Seja R o número máximo de retransmissões que podem ser feitas no nível de transporte.

Observe que a utilização de um único ACK fim-a-fim para a recuperação de uma eventual perda, estará sujeita às vulnerabilidades acumuladas de toda a rota de ida e volta. Assim, para o sucesso na transmissão fim-a-fim, a mensagem terá que atravessar os h saltos da origem (nó fonte) ao destino (nó sorvedouro) e o ACK terá que atravessar os mesmos h saltos de volta (supondo que não houve alteração de rota). Em caso de falha, qualquer uma das R retransmissões (novas tentativas) ocorrerão mais uma vez fim-a-fim. Assim, a probabilidade de sucesso na entrega de uma mensagem, encapsulada em um único pacote, à estação base, h saltos distante do nó origem, será:

p^h : sucesso em todo o caminho de ida (MSG)

q^h : sucesso em todo o caminho de volta (ACK)

$p^h\, q^h$: sucesso ida e volta

$P(f) = (1 - p^h\, q^h)$: probabilidade de alguma falha ao longo de todo o trajeto

$P(f)_R = (1 - p^h\, q^h)^{R+1}$: probabilidade de falha em todas as transmissões

Assim, a equação 1 nos dá a probabilidade de sucesso com confirmação (ACK) fim-a-fim e com até R retransmissões

$$P(s)_{ACK\ fim-a-fim} = 1 - (1 - p^h\, q^h)^{R+1} \tag{1}$$

Conforme descrito na seção 3.2, o protocolo proposto propõe a recuperação de falhas feita salto-a-salto, com ACKs enviados por cada vizinho ao nó anterior. Deste

modo, a análise probabilística passa a ser a seguinte:

Probabilidade de sucesso $P(s)$ para um único salto:

$$P(s) = \sum_{i=0}^{R-1} pq\,(1-pq)^i \tag{2}$$

A expressão correspondente, para a probabilidade de não falhar nas R retransmissões, é:

$$P(s) = 1 - (1-pq)^R \tag{3}$$

Assim, a probabilidade de sucesso na entrega de uma mensagem, encapsulada em um único pacote, à estação base, h saltos distante do nó origem, será:

$$P(s)_{ACK\ salto-a-salto} = (P(s))^h \tag{4}$$

Observamos pelas equações 1 e 4 e pela figura 5, que a recuperação salto-a-salto oferece uma garantia de entrega muito maior que o esquema com recuperação fim-a-fim. As equações foram resolvidas para $R = 3$ baseado em [Stann and Heidemann 2003].

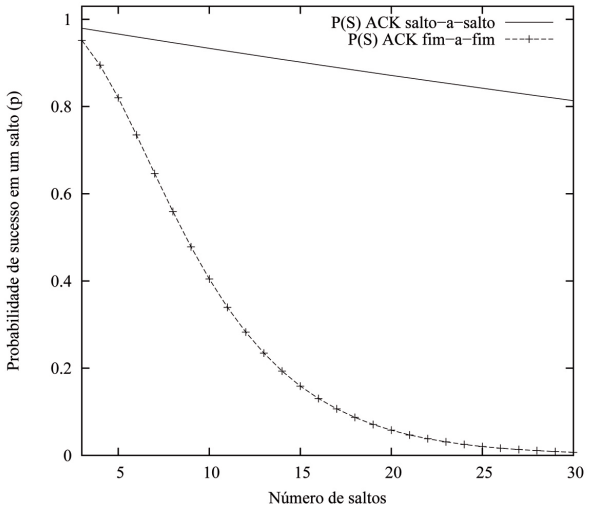


Figura 5. Probabilidade de entrega fim-a-fim, com ACK fim-a-fim e ACK salto-a-salto. ($R = 3$)

3.3.2. Análise da falha dos nós

Como vimos na seção 3.3.1, as falhas de comunicação podem ser tratadas através do envio de ACKs e posterior retransmissão corretiva de dados. Um problema adicional reside na possível falha do nó que contém a informação a ser transmitida.

No caso da transmissão com ACK fim-a-fim, a informação se mantém armazenada apenas no nó origem, até que um ACK informe que a mesma chegou ao destino. Um

defeito, dano ou esgotamento de energia neste nó antes que a mensagem chegue ao destino ocasionará sua perda definitiva.

No transporte com nível 1 de confiabilidade, a mensagem se mantém armazenada sucessivamente nos nós intermediários, até que o nó subseqüentes envie o ACK, assumindo a responsabilidade de mantê-la até que o próximo salto seja concluído. Mais uma vez, uma falha no nó detentor da informação causará a perda definitiva da mesma.

No transporte com nível 2 de confiabilidade, o armazenamento é feito por 2 nós adjacentes ao mesmo tempo, o que reduz a probabilidade de perda da mensagem. Apenas a falha destes dois nós causa a perda da informação.

Consideremos que a probabilidade de um nó **não** falhar é dada por f. A perda total de uma mensagem se dará caso as falhas de nós citadas acima ocorram em intervalos críticos.
Considere os nós A, B e C representados nas figuras 2 e 4.

Os intervalos críticos da confiabilidade no nível 1 são:

- B recebe a MSG de A com sucesso, envia o ACK para A e falha antes de enviar a MSG para C.
- B recebe a MSG de A com sucesso, envia o ACK para A e envia a MSG para C. A MSG falha e não chega em C. Antes de retransmitir B falha.

Observe ainda que a seqüência B_FALHOU E ACK_BA_FALHOU não implica na perda definitiva de MSG, já que A ainda não descartou MSG.

Os intervalos críticos da confiabilidade nível 2, são:

- Considere a figura 4. A perda definitiva da mensagem se dará se e somente se os nós A, B e C falharem, juntos, antes que C tenha enviado a MSG ao nó D. Ou seja, $ACK2\_BA\_OK\ E\ FALHA$ de **todos** os nós que já receberam MSG antes do último nó na cadeia enviar a MSG ou após este envio, mas com falha do mesmo.

A rigor, basta uma cópia de MSG ainda não reconhecida pela segunda vez (ACK2), ou seja, basta existir um nó que tenha MSG e que ainda não tenha recebido o ACK2 correspondente, para que MSG ainda não tenha sido perdida em definitivo.

Aqui identificamos o conceito de elasticidade do protocolo. Uma vez enviada pela origem, MSG se propaga indefinidamente (até o destino no máximo). Dependendo do sucesso de envio e recebimento das mensagens ACK2, haverá um maior ou menor consumo distribuído de buffer para uma mesma MSG. A elasticidade é boa, na medida em que pode gerar mais e mais cópias de MSG (mais uma vez dependendo do sucesso de envio e recebimento das mensagens ACK2), o que reduz a possibilidade de perda definitiva de MSG. O mais provável é que, de um modo geral, o protocolo se comporte da maneira esperada, ou seja, com 2 cópias de MSG na rede a cada instante.

Análise das falhas de nós na confiabilidade 1

Seja $f(t)$ a propabilidade de um nó falhar em um intervalo de tempo t

Seja $t1 =$ Intervalo crítico na confiabilidade 1

Seja $P_1(F)$ a probabilidade de perda definitiva de MSG na confiabilidade 1

Temos então: $P_1(F) = f(t1)$

Análise das falhas de nós na confiabilidade 2

Seja k o grau de elasticidade do protocolo.

k = número de nós subseqüentes ao mais antigo detentor de MSG que já receberam MSG.
(k varia com todas as caracteríticas momentâneas da rede, acesso ao meio, contenção, condições climáticas, densidade de nós, tráfego etc).

Seja $t2$ = Intervalo crítico na confiabilidade 2

Seja $P_2(F)$ a probabilidade de perda definitiva de MSG na confiabilidade 2

Temos então: $P_2(F) = (f(t2))^k$

Como $t1$ e $t2$ se referem a um instante entre um envio de um ACK e o envio com sucesso de uma MSG, pode-se considerar $t1 = t2$. Sendo a chance de sucesso no envio de um ACK aproximadamente a mesma que o envio de uma MSG, tem-se $k = 2$.

Logo: $P_2(F) = (f(t2))^2 = (f(t1))^2 = (P_1(F))^2$

Assim, a probabilidade de perda definitiva da mensagem é consideravelmente menor no esquema com confiabilidade 2, como vemos na figura 6.

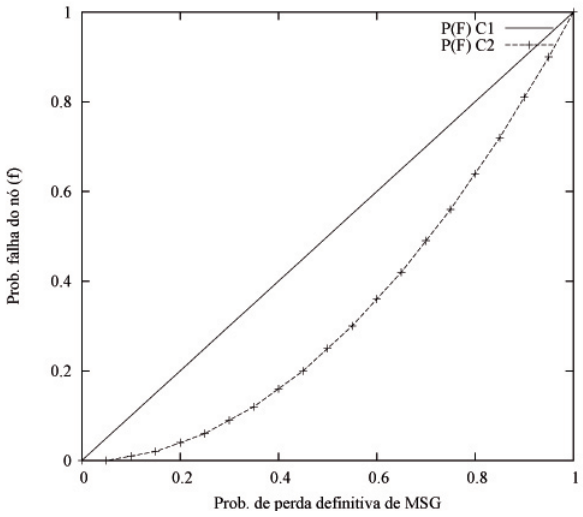


Figura 6. Probabilidade de perda definitiva da mensagem nos dois níveis de confiabilidade

3.4. Detecção e controle de congestionamento

Detecção e controle de congestionamento em redes de sensores, é um assunto importante. O mecanismo de detecção antecipada e randômico usado em redes tradicionais propõe que um nó intermediário descarte um pacote quando existe congestionamento na rede. A origem perceberá o descarte através de um ACK ou NACK. Contudo, descartar pacotes em redes de sensores não é uma solução aceitável.[Iyer et al. 2005]

Portanto, foi preciso considerar outras opções. Nas redes de sensores, as perdas de pacotes normalmente se referem a erros de transmissão e não a congestionamentos. Por isso, qualquer perda de pacote inicializaria o mecanismo de controle de congestionamento e reduziria a taxa de transmissão sem necessidade. Assim, faz-se necessária a implementação de um mecanismo de controle de congestionamento que saiba diferenciar uma perda de pacote relativa a esgotamento de *buffers* de um perda de pacote relativa a erro de transmissão.

Nesta proposta o controle de congestionamento é implementado através da participação de todos os nós intermediários na conexão de transporte. Estes nós gerenciam o congestionamento utilizando mensagens de sinalização. Desta forma, um nó se recusa a receber mais pacotes, caso seus *buffers* estejam ocupados. Logo, quando os *buffers* do nó B atingem o patamar T, este nó envia um pacote sinalizador (*STOP*), em *broadcast*, para todos os seus vizinhos, avisando que pacotes não podem mais ser enviados para ele pois seus *buffers* estão sem espaço de armazenamento. Tal atitude diminuirá a taxa de transmissão de seus vizinhos e conseqüentemente dos vizinhos dos vizinhos. Essa redução na taxa de transmissão poderá se propagar por toda a rede, dependendo do nível de congestionamento existente no momento.

Quando o nó conseguir esvaziar os seus *buffers*, ou seja, quando eles estiverem abaixo do patamar T, um novo pacote sinalizador (*START*) é enviado para liberar o encaminhamento de novos pacotes. Este patamar será calculado, a princípio, empiricamente, através de testes realizados. Portanto, consegue-se, através do mecanismo descrito nesta seção, concluir que toda perda de pacotes, será decorrente de erros de transmissão e não de congestionamento.

4. Conclusões e trabalhos futuros

Neste trabalho, nós apresentamos o Protocolo de Transporte Colaborativo para Rede de Sensores (CTCP). Este protocolo promove entrega confiável de mensagens entre um nó e sua respectiva estação base. Ele é capaz de detectar e controlar o congestionamento através da diferenciação entre as perdas relativas a erro de transmissão e aquelas geradas por esgotamento de *buffers*. Em caso de perdas, ele oferece recuperação em dois níveis de confiabilidade e, em caso de congestionamento, oferece sinalização explícita de interrupção e retorno do envio de dados.

Analisamos probabilisticamente o mecanismo de confiabilidade proposto e concluímos que a recuperação distribuída de erros (salto-a-salto) aumenta significativamente a probabilidade de entrega de mensagens fim-a-fim. Observamos ainda que, dividindo a responsabilidade de armazenamento temporário da mensagem entre dois nós adjacentes (nível 2 de confiabilidade), temos uma queda importante da probabilidade de perda definitiva de uma mensagem. Além disso, o armazenamento distribuído de mensagens até que

um ACK confirme que a mesma já se encontra sob responsabilidade do próximo (ou dos 2 próximos) nó(s) demonstra a robustez do protocolo diante de períodos de desconexão.

O CTCP não faz qualquer restrição quanto aos protocolos de níveis inferiores que serão usados.

A estação base assume as decisões que dependem do conhecimento dos requisitos da aplicação (confiabilidade requerida) e controla parâmetros com características centrais (ID da conexão). Por outro lado, as funções de controle de congestionamento, implementação da confiabilidade e abertura de conexões estão distribuídas na RSSF.

Em trabalhos futuros pretendemos investigar o efeito do aumento do número de saltos na geração de ACKs múltiplos no modelo de nível 2 de confiabilidade. Utilizaremos como métrica o aumento do custo de transmissão e de consumo de *buffers* versus aumento na probabilidade de entrega.

Através da implementação deste protocolo, em um ambiente de simulação, pretendemos analisar a multiplexação de fluxos (ou conexões) de transporte.

Referências

Allman, M., Paxson, V., and Stevens, W. (1999). Tcp congestion control. RFC 2581 (Proposed Standard). Updated by RFC 3390.

C. Intanagonwiwat, RC. Govindan, D. E. (2000). Direct diffusion: A scalable and robust communication paradigm for sensor networks. In *In Proceedings of the Sixth Annual ACM International Conference on Mobile Computing and Networking*, pages 56–67.

Iyer, Y. G., Gandham, S., and Venkatesan, S. (2005). Stcp: A generic transport layer protocol for wireless sensor networks. In *Proceedings of 14th International Conference on Computer Communications and Networks*, pages 449–454, Houston, TX, USA.

Kim, S., Fonseca, R., Dutta, P., Tavakoli, A., Culler, D., Levis, P., Shenker, S., and Stoica, I. (2007). Flush: a reliable bulk transport protocol for multihop wireless networks. In *SenSys '07: Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 351–365, New York, NY, USA. ACM.

Milner, R. (1980). *A Calculus of Communicating Systems*. Springer Verlag.

Nielsen, M., Plotkin, G., , and Winskel, G. (1981). *Petri Nets, Event Structures and Domains, Part I, Vol. 13*. Theoretical Computer Science.

Sankarasubramaniam, Y., Akan, O., and Akyildiz, I. (2003). Esrt: Event-to-sink reliable transport in wireless sensor networks. In *In Proceedings of MobiHoc 03, ACM*, Annapolis, Maryland, USA.

Stann, F. and Heidemann, J. (2003). Rmst: Reliable data transport in sensor networks. In *Proceedings of the First International Workshop on Sensor Net Protocols and Applications*, pages 102–112, Anchorage, Alaska, USA.

Vassis, D., Kormentzas, G., Rouskas, A. N., and Maglogiannis, I. (2005). The ieee 802.11g standard fo high data rate wlans. *IEEE Network*, 19(3):21–26.

Wan, C.-Y., Campbell, A. T., and Krishnamurthy, L. (2005). Pump-slowly, fetch-quickly (psfq): A reliable transport protocol for sensor networks. In *IEEE Journal on Selected Areas in Communications, Vol. 23, No. 4*, pages 862–872, Atlanta, Georgia, USA.

Consenso Bizantino entre Participantes Desconhecidos*

Eduardo A. P. Alchieri[1], Alysson N. Bessani[2], Joni S. Fraga[1], Fabíola Greve[3]

[1]DAS, Universidade Federal de Santa Catarina, Florianópolis

[2]LaSIGE, Faculdade de Ciências da Universidade de Lisboa, Lisboa

[3]DCC, Universidade Federal da Bahia, Bahia

Resumo. *O problema do consenso é a base para a solução da maioria dos problemas que envolvem sistemas distribuídos confiáveis. Apesar do consenso estar amplamente estudado em ambientes clássicos, onde o conjunto de participantes é conhecido, poucos trabalhos consideram este problema em ambientes dinâmicos e auto-organizáveis, onde os participantes da computação são, a priori, desconhecidos. Neste trabalho, propomos duas soluções para o consenso bizantino entre participantes desconhecidos (BFT-CUP), com e sem o uso de assinaturas digitais, e provamos que o grau de conhecimento sobre os participantes do sistema necessário nestas soluções é o mínimo necessário para resolver o BFT-CUP.*

1. Introdução

O problema do consenso [Lamport et al. 1982, Chandra and Toueg 1996], e de um modo mais geral os algoritmos de acordo, formam a base para a solução da maioria dos problemas encontrados no desenvolvimento de sistemas distribuídos confiáveis, pois possibilitam que os participantes da computação distribuída coordenem suas ações de forma a manter a consistência em seus estados e garantir o progresso do sistema. Primeiramente, este problema foi estudado em redes clássicas, onde o conjunto de processos que participam de determinada computação é estático e conhecido por todos os participantes do sistema. Mesmo nestes ambientes, várias dificuldades são encontradas na solução deste problema, que não pode ser resolvido de forma determinista em um sistema assíncrono com possibilidade de falhas [Fischer et al. 1985].

Quando consideramos um sistema dinâmico e auto-organizável, como redes *ad-hoc*, redes de sensores e, num contexto diferente, redes entre pares (P2P) não estruturadas, as dificuldades encontradas na elaboração de protocolos para resolver o problema do consenso aumentam significativamente, pois nestes ambientes os participantes da computação são, *a priori*, desconhecidos (tanto a número de participantes quanto suas identidades). Desta forma, não é possível o simples emprego de protocolos de consenso estático neste novo ambiente, pois tais protocolos consideram que os participantes do sistema são, *a priori*, conhecidos.

Sendo assim, os primeiros esforços para resolver o consenso em redes desconhecidas (ou sistemas dinâmicos e auto-organizáveis) surgiram em iniciativas como o CUP (*Consensus with Unknown Participants*) [Cavin et al. 2004], que têm por objetivo resolver o consenso em redes desconhecidas, considerando que os processos não podem falhar. Outras iniciativas, como o FT-CUP (*Fault-Tolerant* CUP) [Cavin et al. 2005, Greve and Tixeuil 2007], buscam estender o modelo anterior fornecendo suporte a falhas por parada (*crash*) dos participantes do sistema.

Neste trabalho, definimos uma nova versão do problema de consenso em redes desconhecidas, chamada de BFT-CUP (*Byzantine Fault-Tolerant* CUP), onde o problema do consenso em redes desconhecidas é resolvido considerando que os participantes do sistema podem se comportar de forma maliciosa [Lamport et al. 1982]. Deste modo, além de tratar as questões inerentes aos sistemas auto-organizáveis, os protocolos devem tolerar a possibilidade de participantes maliciosos estarem presentes no sistema (o que é bastante factível, dada sua natureza aberta), tentando impedir que o consenso seja atingido. Duas soluções para o BFT-CUP são

*Alchieri é bolsista CAPES; Fraga e Greve são bolsistas CNPq.

propostas neste artigo, com e sem o uso de assinaturas digitais, requerendo diferentes graus de conectividade do sistema. Além disso, é provado que estas soluções são ótimas em termos da conectividade requerida entre os participantes do sistema neste modelo faltas, quando assumimos o mínimo de sincronia necessária para resolver consenso.

2. Modelo de Sistema

Assume-se o modelo de sistema com sincronia parcial: existem limites para o tempo necessário para a transmissão de uma mensagem e para a realização de qualquer computação que terminam por valer no sistema, no entanto, esses limites não são conhecidos [Dwork et al. 1988]. A idéia por trás deste modelo é de que o sistema trabalha de forma assíncrona (não respeitando nenhum limite de tempo) a maior parte do tempo. Porém, durante períodos de estabilidade, o tempo para transmissão de mensagens é limitado. Este modelo é necessário para a execução de um protocolo de consenso bizantino clássico ou estático (seção 3.4.4).

Em relação aos processos, considera-se um sistema distribuído formado por um conjunto finito Π de $n > 1$ processos[1], sendo que cada processo $p_i \in \Pi$ conhece apenas um subconjunto Π_i de Π. As comunicações entre estes processos se dão através do algoritmo definido na seção 3.3. Um processo p_i apenas pode enviar mensagens para outro processo p_j se $p_j \in \Pi_i$. Além disso, se p_i envia uma mensagem para p_j, tal que $p_i \notin \Pi_j$, então, após p_j receber a mensagem o mesmo pode adicionar p_i em Π_j, i.e., p_j passa a poder enviar mensagens para p_i. Implicitamente, considera-se que existe uma camada de roteamento tolerante a faltas bizantinas [Castro et al. 2002, Awerbuch et al. 2002] responsável por encaminhar as mensagens entre os processos que se conhecem. Processos do sistema estão sujeitos a falhas bizantinas [Lamport et al. 1982]: um processo que apresenta este tipo de falha pode exibir qualquer comportamento, podendo parar, omitir envios ou entregas de mensagens, ou desviar arbitrariamente de sua especificação. Um processo que apresenta comportamento de falha é dito falho, de outra forma é dito correto. O número máximo de processos que podem falhar f é conhecido por todos os processos do sistema.

3. BFT-CUP: Consenso Bizantino entre Participantes Desconhecidos

Em um sistema distribuído, formado por vários processos independentes, o problema do consenso consiste em fazer com que todos os processos corretos acabem por decidir o mesmo valor, o qual deve ter sido previamente proposto por algum dos processos do sistema. Formalmente, o consenso pode ser definido pelas seguintes propriedades [Castro and Liskov 2002, Chandra and Toueg 1996]: *Validade*: um processo correto decide pelo valor v somente se v foi previamente proposto por algum processo; *Acordo*: se um processo correto decide pelo valor v, então todos os processos corretos terminam por decidir v; *Terminação*: todos os processos corretos terminam por decidir; *Integridade*: cada processo correto decide no máximo uma vez.

Nesta seção são apresentadas duas soluções para o BFT-CUP. A primeira utiliza assinaturas digitais e necessita de uma menor conectividade no grafo que representa o conhecimento dos participantes. A segunda, que não faz uso de assinaturas digitais, requer um maior grau de conhecimento sobre os participantes que estarão presentes em uma dada execução do consenso. As principais diferenças entre as duas abordagens estão relacionadas com a comprovação da autenticidade das mensagens enviadas pelos participantes (seção 3.3) e com a fase de descoberta de participantes (seção 3.4.2). Além disso, quando assinaturas digitais são empregadas, torna-se necessário a existência de uma autoridade certificadora[2], reconhecida por todos os processos,

[1]Neste trabalho os termos nós, vértices, processos e participantes serão usados indistintamente.

[2]Não é necessário que esta autoridade certificadora esteja *online* durante a execução dos protocolos, mas sim na criação dos certificados (e suas possíveis revogações). Somente é necessário que os participantes conheçam a chave pública desta autoridade, a fim de validar os certificados emitidos pela mesma.

responsável pelo gerenciamento das chaves dos participantes do BFT-CUP. Assim, cada participante do sistema possui um par de chaves pública-privada, sendo a chave privada conhecida apenas pelo próprio participante. Já as chaves públicas de todos os participantes são conhecidas por todos os outros participantes (através de certificados).

As principais ações que um participante malicioso pode executar para prejudicar a realização de BFT-CUP são: (*i.*) omitir o conhecimento sobre determinado(s) participante(s); (*ii.*) inventar a existência de participantes ou se fazer passar por outros participantes; (*iii.*) mandar dados diferentes para cada participante, tentando impedir que as propriedades do consenso sejam atendidas. Diante disto, as soluções para o BFT-CUP devem mascarar estas ações executadas por participantes maliciosos. As seções seguintes apresentam os protocolos propostos.

3.1. Detectores de Participação

É impossível resolver o consenso caso cada processo tenha conhecimento apenas sobre si próprio. Desta forma, as informações que um processo obtém sobre os demais participantes do sistema são capturadas através da noção de detectores de participação [Cavin et al. 2004], os quais são oráculos distribuídos que fornecem "dicas" aos processos sobre quais destes estão presentes na computação distribuída. Seja $i.PD$ o detector de participação do processo i, uma consulta a $i.PD$ retorna um subconjunto de processos de Π com os quais i pode colaborar, i.e., os vizinhos de i. Sendo $i.PD(t)$ o resultado de uma consulta de i no tempo t, a informação retornada por $i.PD$ pode evoluir entre consultas, mas segue as seguintes propriedades [Cavin et al. 2004]: *Inclusão da Informação*, a informação retornada pelo detector de participação não diminui com o tempo, i.e., $\forall i \in \Pi, \forall t' \geq t : i.PD(t) \subseteq i.PD(t')$; *Exatidão da Informação*, os detectores de participação não cometem erros, i.e., $\forall i \in \Pi, \forall t : i.PD(t) \subseteq \Pi$.

Os detectores de participação fornecem aos processos um contexto inicial sobre os participantes do sistema, através do qual é possível expandir o conhecimento sobre Π. Deste modo, cada participante obterá um grafo de conectividade por conhecimento que é orientado, pois a relação de conhecimento não é necessariamente bidirecional [Cavin et al. 2004].

Definição 1 *Grafo de conectividade por conhecimento: Seja $G_{di} = (V, \xi)$ um grafo orientado representando a relação de conhecimento induzida por um detector de participação PD. Então, $V = \Pi$ e $(i, j) \in \xi$ sse $j \in i.PD$, i.e., i conhece j.*

Definição 2 *Grafo não orientado de conectividade por conhecimento: Seja $G = (V, \xi)$ um grafo não orientado representando a relação de conhecimento induzida por um detector de participação PD. Então, $V = \Pi$ e $(i, j) \in \xi$ sse $j \in i.PD$ ou $i \in j.PD$.*

De acordo com as características observadas nos grafos de conhecimento, algumas classes de detectores de participação foram propostas para resolver o CUP [Cavin et al. 2004] e o FT-CUP [Greve and Tixeuil 2007]. A principal classe abordada na resolução do FT-CUP é:

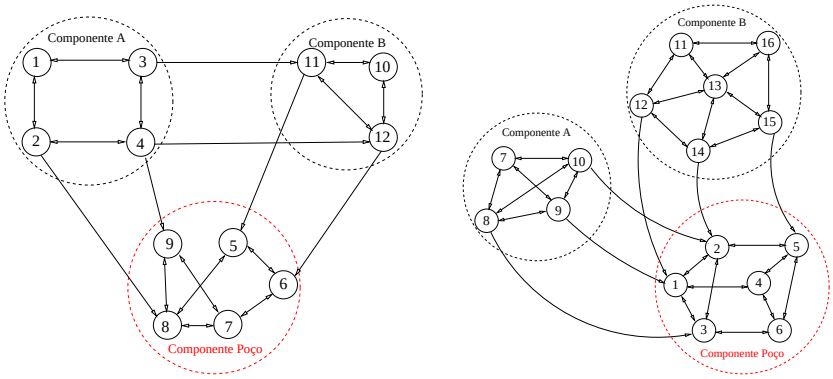
PD *k*-Redutível a Único Poço (*k*-OSR): O grafo *orientado* G_{di}, que representa a relação de conhecimento induzida pelo detector de participação PD, satisfaz as seguintes condições: (*i.*) o grafo de conectividade por conhecimento G, obtido de G_{di}, é *conexo*; (*ii.*) o grafo direcionado acíclico, obtido pela redução de G_{di} às suas componentes *k*-fortemente conexas[3], tem uma e somente uma *componente poço*[4]; (*iii.*) considere quaisquer duas componentes *k*-fortemente conexas G_1 e G_2, se existe um caminho de G_1 para G_2, então existem k *caminhos disjuntos* entre os vértices de G_1 para G_2.

A Figura 1 apresenta dois grafos G_{di} induzidos por detectores de participação pertencentes a classe *k*-OSR. As Figuras 1(a) e 1(b) correspondem a relações de conhecimento induzidas por detectores de participação 2-OSR e 3-OSR, respectivamente. Na solução que utiliza assina-

[3]Uma componente G_{fc} é *k-fortemente conexa* se para qualquer par de vértices (v_i,v_j) de G_{fc}, v_i pode alcançar v_j através de *k* caminhos disjuntos nos vértices.

[4]Uma componente G_p de G_{di} é considerada *poço* quando não existem caminhos saindo de vértices em G_p para vértices pertencentes a outras componentes de G_{di}.

turas, o retorno do detector de participação associado a um participante i é assinado pelos seus vizinhos, i.e., cada participante j assina a informação indicando que é vizinho de i ($\langle j \rangle_{\sigma j}$).



(a) 2-OSR (b) 3-OSR

Figura 1. Grafos de conectividade induzidos por detectores de participação.

3.2. Detector de Participação *k*-Redutível a Única Fonte (*k*-OFR)

Apesar de resolvermos o BFT-CUP através do PD k-OSR (seção 3.4), esta seção introduz uma nova classe de detectores de participação, chamada **PD *k*-Redutível a Única Fonte (*k*-OFR)**, assim definida: O grafo de conectividade por conhecimento G_{di} satisfaz as mesmas propriedades definidas para a classe *k*-OSR, sendo que a propriedade (*ii.*) deve ser alterada: (*ii.*) O grafo direcionado acíclico, obtido pela redução de G_{di} às suas componentes *k*-fortemente conexas, tem uma e somente uma *componente fonte*[5].

De acordo com a definição de *equivalência entre detectores de participação* [Cavin et al. 2004], os detectores *k*-OFR e *k*-OSR são equivalentes. Então, qualquer resultado obtido com um detector pode ser aplicado ao outro. A prova desta equivalência é baseada no fato da componente poço de um detector exercer o papel da componente fonte do outro, e vice-versa. Note que, o grafo de conhecimento apresentado na Figura 1(a) pode ser obtido também através de um PD 2-OFR (a componente A é fonte). Assim, o PD *k*-OFR também pode ser usado para resolver o BFT-CUP.

3.3. Protocolo de Disseminação

Esta seção discute como ocorre a comunicação entre os processos participantes do sistema. Duas soluções são propostas (com e sem a utilização de assinaturas digitais), onde consideramos que um canal confiável e autenticado é estabelecido entre os participantes vizinhos, possibilitando o envio de mensagens de forma confiável entre os nós vizinhos. A autenticidade das mensagens é comprovada através do uso ou de assinaturas ou de redundância de mensagens (dependendo da solução adotada), como será explicado a seguir. Além disso, apenas é necessário que este canal seja estabelecido entre os nós vizinhos. Deste modo, um participante não é capaz de mentir sobre sua identidade para um vizinho (canais autenticados), uma premissa fundamental para a realização de algoritmos tolerantes a faltas bizantinas [Castro and Liskov 2002].

A comunicação é feita através de duas primitivas: ***dependable_send(***$message$***,***$sender$***)*** - que faz com que um participante $sender$ envie uma mensagem $message$ para todos os participantes alcançáveis[6] a partir de $sender$, i.e., realiza um *flooding* da mensagem; e ***dependable_deliver(***$message$***,***$sender$***,***$routes$***)*** - chamada no participante receptor para entrega da mensagem $message$ difundida pelo participante $sender$, $routes$ contém as rotas (caminhos) através das quais $message$ foi recebida. Estas primitivas devem satisfazer as seguintes propriedades:

[5]Uma componente G_f de G_{di} é considerada *fonte* quando não existem caminhos saindo de vértices pertencentes a outras componentes de G_{di} para vértices em G_f.

[6]Uma participante q é alcançável por outro participante p se existir um número suficiente de caminhos disjuntos nos nós de p para q. Neste caso, q será um receptor da mensagem difundida por p.

- **Validade**: *Uma mensagem difundida por um participante correto $sender$ sempre acabará por ser entregue por algum participante correto alcançável a partir de $sender$ ou não existe nenhum participante correto alcançável por $sender$;*
- **Acordo**: *Se algum participante correto entregar a mensagem $message$, difundida por um participante correto $sender$, então todos os participantes corretos alcançáveis por $sender$ também entregarão $message$;*
- **Integridade**: *Para qualquer mensagem $message$, cada participante correto i entregará $message$ apenas se esta foi previamente difundida por algum participante $sender$, neste caso i é alcançável a partir de $sender$.*

O algoritmo 1 representa este protocolo de disseminação, onde estas primitivas são implementadas. Em nossas soluções, assumimos que o detector de participação de cada processo é consultado exatamente uma vez, o que contribui para que o grafo de conectividade por conhecimento obtido seja consistente, i.e., o conjunto de vizinhos não sofrerá alterações durante toda a execução do protocolo. Esta premissa pode ser implementada através do armazenamento da primeira consulta ao detector de participação e do retorno deste valor em consultas futuras.

Algoritmo 1 Algoritmo de comunicação executado pelo participante i

```
constant:
(1) f:int //número máximo de faltas suportadas

variables:
(2) i.use_signature:boolean //variável que indica o uso de assinaturas
(3) i.received_msgs:set of <message,route> tuples //conj.  de mensagens recebidas

message:
(4) DEPENDABLE_FLOODING: //estrutura da mensagem DEPENDABLE_FLOODING
(5)     message:value to flood // valor a ser difundido
(6)     route:ordered list of nodes //caminho percorrido pela mensagem

** Initiator Only **
procedure:
dependable_send(message,sender) // sender == i
(7)     if i.use_signature then
(8)         message = ⟨message⟩_{σ_i}; //i assina a mensagem
(9)     end if
(10)    for each j ∈ i.PD do
(11)        SEND DEPENDABLE_FLOODING(message,i) to j;
(12)    end for

** All Nodes **
INIT:
(13) i.use_signature = true/false; i.received_msgs = ∅;

(14) upon receipt of DEPENDABLE_FLOODING(m.message,m.route) from j
(15)    if getLastElement(m.route) == j ∧ i ∉ m.route then
(16)        addAtEnd(m.route,i);
(17)        initiator = getFirstElement(m.route);
(18)        if i.use_signature then
(19)            if verify_signature(⟨m.message⟩_{σ_initiator}) then
(20)                dependable_deliver(m.message,initiator,m.route);
(21)                for each z ∈ i.PD \ {j} do
(22)                    SEND DEPENDABLE_FLOODING(m.message,m.route) to z;
(23)                end for
(24)            end if
(25)        else
(26)            i.received_msgs = i.received_msgs ∪ {⟨m.message,m.route⟩};
(27)            routes = computeRoutes(m.message,i.received_msgs);
(28)            if (#⟨m.message,*⟩ ∈ i.received_msgs ≥ f + 1) ∧ (routes ≥ f + 1) then
(29)                dependable_deliver(m.message,initiator,routes);
(30)                i.received_msgs = i.received_msgs \ {⟨m.message,*⟩};
(31)            end if
(32)            for each z ∈ i.PD \ {j} do
(33)                SEND DEPENDABLE_FLOODING(m.message,m.route) to z;
(34)            end for
(35)        end if
(36)    end if
```

A idéia principal deste algoritmo é que os participantes façam *flood* de suas mensagens, atingindo todos os processos que podem alcançar. Desta forma, um participante que deseja difundir uma mensagem deve executar o procedimento $dependable\_send$. Neste procedimento, a mensagem é enviada a todos os seus vizinhos (linhas 10-12), caracterizando um

flooding, onde a mensagem recebida de um vizinho é encaminhada para os demais vizinhos e assim sucessivamente até alcançar todos os participantes possíveis (linhas 21-23 e 32-34). Uma característica importante desta difusão é o fato de cada mensagem ir acumulando a rota conforme o caminho percorrido até chegar em determinado destino. Um participante apenas considera determinada mensagem quando o processo que a está enviando (ou encaminhando) estiver adicionado no final da rota acumulada (linha 15). Esta solução é baseada na abordagem apresentada em [Dolev 1982], que impossibilita um participante malicioso de não se adicionar no final das informações de rota durante o envio (ou encaminhamento) de uma mensagem.

No entanto, um participante malicioso pode retirar ou adicionar participantes na rota acumulada (sejam eles corretos ou não), modificar a mensagem que está sendo propagada e até mesmo não retransmitir a mesma. Assim, é necessário que a conectividade do grafo que representa o conhecimento dos participantes seja suficientemente grande para fazer com que a mensagem chegue aos destinatários e que os mesmos possam provar sua autenticidade, possibilitando que o protocolo continue funcionando como especificado. Além disso, esta abordagem impossibilita que um participante malicioso invente mensagens que não foram difundidas por outros participantes, pois a autenticidade da mesma não será comprovada (ver adiante).

Conforme já comentado, para que um participante seja capaz de difundir suas mensagens apropriadamente, o grau de conectividade k do grafo que representa o conhecimento dos participantes deve ser suficientemente grande e varia de acordo com o uso ou não de assinaturas. Além disso, como as provas de corretude do algoritmo 1 diferem entre as duas soluções, estas abordagens serão estudadas separadamente.

Com assinaturas. Nesta abordagem, para que um participante i seja capaz de alcançar um participante p, devem existir no mínimo $f + 1$ caminhos disjuntos nos nós de i para p, i.e., o grau de conectividade k deve ser assumido como $k > f$. As linhas 19-24 do algoritmo 1 representam o processamento quando assinaturas são utilizadas. Assim, quando uma mensagem é recebida por algum participante p e sua assinatura é verificada (linha 19), comprovando a autenticidade da mesma, tal mensagem é entregue por p (linha 20), que também a encaminha para seus vizinhos (linhas 21-23).

Provas de Correção. *Validade*: Esta solução requer $k > f$. Desta forma, teremos no mínimo $f + 1$ caminhos disjuntos nos nós entre o emissor e os receptores (nós alcançáveis pelo emissor) de uma mensagem. Sendo assim, existe no mínimo um caminho formado apenas por participantes corretos, através do qual é garantido que a mensagem chegará aos receptores, os quais detectarão a autenticidade da mesma através da assinatura e a entregarão através da operação $dependable\_deliver$ (linha 20). *Acordo*: Como o acordo é definido sobre mensagens enviadas por participantes corretos, esta prova acaba se tornando idêntica a prova da propriedade de validade. *Integridade*: Como uma mensagem apenas é entregue após sua assinatura ser comprovada (linha 19), não é possível que um participante malicioso forje mensagens, i.e., apenas mensagens autênticas são entregues pelo protocolo. Sendo assim, toda mensagem entregue por algum participante correto foi previamente difundida pelo seu emissor ($sender$). □

Sem assinaturas. Nesta abordagem, para que um participante i seja capaz de alcançar um participante p, devem existir no mínimo $2f + 1$ caminhos disjuntos nos nós de i para p, i.e., o grau de conectividade k deve ser assumido como $k > 2f$. As linhas 26-34 do algoritmo 1 representam o processamento quando assinaturas não são utilizadas. Desta forma, uma mensagem recebida é armazenada em um conjunto (linha 26) e somente é entregue (linha 29) após a mesma mensagem ser recebida $f + 1$ vezes através de $f + 1$ caminhos disjuntos nos nós[7] (linhas 27-28), i.e., ter sua autenticidade comprovada. Além disso, sempre que uma mensagem é recebida por algum participante, o mesmo a encaminha para seus vizinhos (linhas 32-34).

[7] A função $computeRoute(m.message, i.received\_msgs)$ calcula o número de caminhos (rotas) disjuntos nos nós através dos quais $m.message$ foi recebida.

Provas de Correção. *Validade*: Esta solução requer $k > 2f$. Neste caso, teremos no mínimo $2f + 1$ caminhos disjuntos nos nós entre o emissor e os receptores (nós alcançáveis pelo emissor) de uma mensagem. Então, existe no mínimo $f + 1$ caminhos disjuntos nos nós formados apenas por participantes corretos, através dos quais é garantido que a mensagem chegará aos receptores, que detectarão a autenticidade da mesma através da redundância, i.e., do recebimento de $f + 1$ mensagens iguais através destes $f + 1$ caminhos disjuntos nos nós. Assim, após a autenticidade da mensagem ser comprovada, a mesma é entregue através da operação $dependable\_deliver$ (linha 29). *Acordo*: Como o acordo é definido sobre mensagens enviadas por participantes corretos, esta prova acaba se tornando idêntica a prova da propriedade de validade. *Integridade*: Uma mensagem apenas é entregue após ser recebida $f + 1$ vezes através de $f + 1$ caminhos disjuntos nos nós (linhas 27-28), o que garante que tal mensagem é autêntica, i.e., realmente foi enviada pelo emissor ($sender$). Desta forma, um participante malicioso i não é capaz de inventar que uma mensagem foi enviada por outro participante j (forjar emissor), pois a autenticidade da mesma não será comprovada. Isto é, um receptor r não será capaz de obter os $f + 1$ caminhos disjuntos nos nós de j para r. Mesmo com um conluio de até f participantes maliciosos, r obterá no máximo f caminhos disjuntos nos nós através dos quais recebeu a mensagem "enviada" por j. □

Uma característica encontrada em ambas as soluções é que uma mesma mensagem, enviada por determinado participante, pode ser entregue mais de uma vez pelos seus receptores. Esta característica não prejudica o uso deste protocolo em nosso algoritmo de consenso[8] (seção 3.4). Assim, não tratamos esta limitação no protocolo de comunicação desenvolvido. No entanto, esta entrega duplicada pode ser resolvida através do uso de *buffers* de armazenamento de mensagens já entregues juntamente com a adição de identificadores únicos nas mensagens.

3.3.1. Respondendo às Difusões

Considerando que as mensagens difundidas sejam requisições, e como os caminhos (rotas) percorridos pela mensagem (requisição) em uma determinada difusão são recebidos nos receptores ($dependable\_deliver$), é possível que os mesmos respondam a esta requisição enviando uma mensagem (resposta) pelos caminhos contrários (inversos) ao que recebeu tal requisição. Para isso, é necessário que o receptor de uma requisição descubra $f + 1$ (com assinaturas) ou $2f + 1$ (sem assinaturas) rotas através das quais uma requisição foi recebida, possibilitando que o envio da resposta seja completado mesmo na presença de até f processos faltosos. Deste modo, o grau de conectividade do grafo de conhecimento deve aumentar em f, i.e., $2f + 1$ (com assinaturas) ou $3f + 1$ (sem assinaturas). Assim, um recepetor deverá esperar por $f + 1$ rotas quando utiliza assinaturas (linha 19) ou $2f + 1$ rotas na solução sem assinaturas (linha 28). Note que, esta conectividade apenas é necessária para possibilitar que um receptor responda a uma difusão. Esta abordagem é utilizada em nossos protocolos (seção 3.4), que requerem este grau de conectividade mesmo que os receptores não necessitem respoder à difusões (seção 3.4.2).

Duas primitivas definem o envio da resposta: ***dependable_reply_send(***$message$***,***$sender$***,***$routes$***)*** - usada pelo receptor de uma requisição ($sender$), que responde fazendo com que $message$ percorra os caminhos ($routes$) inversos; e ***dependable_reply_deliver(***$message$***,***$sender$***)*** - chamada no receptor da resposta $message$ (difusor da requisição), para entrega da mesma, que foi enviada por $sender$ (receptor da requisição). A implementação destas primitivas segue os mesmos moldes da implementação da difusão (algoritmo 1), obdecendo às mesmas propriedades. No entanto, é mais simples pelo fato das rotas já estarem predefinidas.

[8]Para que protocolos suportem esta característica, basta considerar apenas a primeira mensagem recebida em uma dada difusão, como é o caso dos nossos algoritmos (seção 3.4).

3.4. Resolvendo o BFT-CUP com o Detector de Participação *k*-OSR

Esta seção apresenta um protocolo para a resolução do BFT-CUP através de um detector de participação *k*-OSR. Como comentado na seção anterior, nossos protocolos requerem $3f+1$ (ou $2f+1$ com assinaturas) caminhos disjuntos nos nós para a descoberta dos participantes (seção 3.4.2). Assim, primeiramente é provado que esta conectividade e este detector de participação são necessários para resolver o BFT-CUP, sendo que suas suficiências são comprovadas através das provas de corretude dos algoritmos apresentados.

Teorema 1 *O detector de participação PD, com $PD \in k\text{-}OSR$ ou equivalente*[9]*, é necessário para resolver o BFT-CUP, dado que f nós podem falhar, onde $f < \frac{k}{2} < n$ (usando assinaturas) ou $f < \frac{k}{3} < n$ (não usando assinaturas).*

Prova: Considere por contradição que existe um algoritmo que resolva o BFT-CUP através de um detector de participação $PD \notin k\text{-}OSR$. Seja G_{di} o grafo de conhecimento induzido por PD, dois cenários são possíveis: (*i.*) ou existem menos de k caminhos disjuntos nos nós conectando um participante p em G_{di} ou (*ii.*) a decomposição de G_{di} em suas componentes *k*-fortemente conexas resulta em mais de um poço. No primeiro cenário, a falha de f nós impossibilitará que p descubra de forma correta os demais participantes do sistema. Esta prova é simples e pode ser assim entendida: considere uma consulta realizada por p em um conjunto de n_c processos, p deve aguardar por $n_c - f$ respostas (pois f processos podem falhar), no entanto, as respostas dos f faltosos podem estar no conjunto de respostas recebidas, assim, é garantido que p recebeu respostas de $n_c - 2f$ processos corretos, sendo que para a consulta ser corretamente realizada é necessário a presença de pelo menos um processo correto caso assinaturas sejam usadas ($n_c - 2f \geq 1$) ou $f + 1$ processos corretos quando não usamos assinaturas ($n_c - 2f \geq f + 1$). Assim, em ambas as situações, chega-se a uma contradição. No segundo cenário, sejam G_1 e G_2 duas das componentes poço e considerando que os participantes em G_1 têm valor de proposição v e os em G_2 valor w, sendo $v \neq w$. Pela propriedade de *terminação* do consenso, os nós em G_1 decidem em um tempo t_1 e os nós em G_2 em um tempo t_2. Atrasando o tempo de recepção de mensagens provenientes de outras componentes para todos os nós em G_1 e G_2 para um tempo $t > max\{t_1, t_2\}$. Como os nós nas componentes poço não sabem da existência de outros nós, pela propriedade da *validade* do consenso, os nós em G_1 decidem pelo valor v e os nós em G_2 pelo valor w, violando a propriedade de *acordo* do consenso ($v \neq w$), chegando-se a uma contradição. □

3.4.1. Visão Geral do Algoritmo

O protocolo para resolução do BFT-CUP tem como base o algoritmo de disseminação apresentado no seção 3.3, que engloba todos os detalhes relacionados com a comunicação entre os participantes do sistema. A partir daí, inspirando-se em [Greve and Tixeuil 2007], o protocolo de consenso é dividido em três fases. Na primeira fase, de *descoberta de participantes* (seção 3.4.2), cada participante do sistema aumentará o seu conhecimento a respeito dos outros participantes, descobrindo o número máximo possível de participantes que estão presentes em determinada computação. A segunda fase, de *determinação da componente poço* (seção 3.4.3), tem como objetivo definir quais são os participantes que pertencem à componente poço do grafo de conhecimento gerado a partir de um PD *k*-OSR. Assim, cada participante é capaz de determinar se pertence ou não à componente poço. Na última fase (seção 3.4.4), os membros da componente poço *executam um consenso tolerante a faltas bizantinas* clássico e propagam o valor de decisão aos demais participantes do sistema. Note que, a componente poço deve conter um número de participantes $n_{sink} \geq 3f + 1$, necessário para a resolução de algum algoritmo de consenso tolerante a faltas bizantinas clássico (ex.: Paxos Bizantino [Castro and Liskov 2002]).

[9] A seção 3.2 apresenta o detector de participação redutível a uma única fonte (*k*-OFR), juntamente com sua equivalência ao detector *k*-OSR. Assim, ambos podem ser empregados na solução do BFT-CUP.

3.4.2. Descoberta dos Participantes

Antes que qualquer computação possa ser iniciada, é necessário que os participantes do sistema obtenham o máximo de conhecimento possível a respeito dos outros participantes. Note que, através de seu detector de participação local, um processo é capaz de determinar quem são seus vizinhos. No entanto, este conhecimento não é suficiente para resolver o BFT-CUP, sendo necessária a sua expansão. Neste sentido, o algoritmo 2 (DISCOVERY) realiza este procedimento. A idéia principal é que cada participante i difunda uma mensagem solicitando informações sobre os vizinhos de cada participante alcançável por i, realizando uma espécie de busca em largura no grafo de conhecimento. No final do algoritmo, i obterá o conjunto maximal de participantes alcançáveis, que representa os participantes conhecidos por i.

Na inicialização do algoritmo em um processo i, o conjunto de conhecidos é atualizado para o próprio i e seus vizinhos e o conjunto de mensagens pendentes (se $j \in i.msg\_pend$, então i deve receber uma mensagem de j) é atualizado para os vizinhos de i (linha 6). Além disso, uma mensagem é difundida para todos os participantes alcançáveis por i (linha 7) solicitando informações a respeito de seus vizinhos. Quando esta mensagem, difundida através do protocolo de disseminação discutido na seção 3.3, é entregue em determinado participante, o mesmo responde para i com seu conjunto de vizinhos (linhas 8-9).

Algoritmo 2 Algoritmo DISCOVERY executado pelo participante i

```
constant:
(1) f:int //número máximo de faltas suportadas

variables:
(2) i.known:set of nodes //conjunto de processos conhecidos por i
(3) i.nei_pend:set of ⟨node,node.neighbor⟩ tuples//i não conhece todos os vizinhos de node
(4) i.msg_pend:set of nodes //conj.  de nós que i espera por mensagens (respostas)
(5) i.use_signature:boolean //variável que indica o uso de assinaturas

** All Nodes **
INIT:
(6) i.known = {i} ∪ i.PD ;i.nei_pend = ∅; i.msg_pend = i.PD;
(7) dependable_send(GET_NEIGHBOR,i);

(8) upon execution of dependable_deliver(GET_NEIGHBOR,sender,routes)
(9)  dependable_reply_send(i.PD,i,routes);

(10) upon execution of dependable_reply_deliver(sender.neighbor,sender)
(11)  i.known = i.known ∪ {sender};
(12)  i.nei_pend = i.nei_pend ∪ {⟨sender,sender.neighbor⟩};
(13)  i.msg_pend = i.msg_pend \ {sender};
(14)  if i.use_signature  then
(15)    for each ⟨j⟩_σj ∈ sender.neighbor do
(16)      if(verify_signature(⟨j⟩_σj) ∨ #_⟨*,⟨j⟩⟩ ∈ i.nei_pend > f ) ∧ j ∉ i.known then
(17)         i.known = i.known ∪ {j};
(18)         i.msg_pend = i.msg_pend ∪ {j};
(19)      end if
(20)    end for
(21)  else
(22)    if  (∃j : #_⟨*,⟨j⟩⟩ ∈ i.nei_pend > f) ∧ j ∉ i.known then
(23)     i.known = i.known ∪ {j};
(24)     i.msg_pend = i.msg_pend ∪ {j};
(25)    end if
(26)  end if
(27)  for each ⟨j,j.neighbor⟩ ∈ i.nei_pend  do
(28)    if  (∀⟨z⟩ ∈ j.neighbor ⟶ z ∈ i.known) then
(29)     i.nei_pend = i.nei_pend \ {⟨j,j.neighbor⟩};
(30)    end if
(31)  end for
(32)  if  (|i.nei_pend| + |i.msg_pend|) ≤ f  then
(33)    return(i.known);;
```

Na computação das respostas em um processo i, o seu conjunto de conhecidos é atualizado, juntamente com os conjuntos de vizinhos pendentes (se $\langle j, j.neighbor\rangle \in i.nei\_pend$, então i conhece j mas não conhece todos os vizinhos de j[10]) e de mensagens pendentes. Além disso, é determinado se i adquiriu conhecimento sobre algum outro participante j (linhas 14-26), i.e., se recebeu esta informação assinada por j ou $f + 1$ outros participantes conhecidos de i informaram que têm j como vizinho. Após estas verificações, o conjunto de vizinhos pendentes é atualizado (linhas 27-30). Para determinar se falta conhecer algum participante, i utiliza

[10]Como i alcança j, i também alcança os vizinhos de j e deve aguardar por suas respostas à difusão inicial de i.

os conjuntos $i.msg\_pend$ e $i.nei\_pend$, que indicam as pendências relacionadas com as mensagens (respostas) recebidas por i (linhas 32-33). O algoritmo termina retornando o conjunto de participantes descobertos por i (linha 33), o qual contém todos os participantes (corretos ou faltosos) alcançáveis a partir de i. Este processamento pode ser entendido como uma busca em largura realizada por i no grafo de conhecimento.

Lema 1 *Seja G_{di} um grafo de conhecimento induzido por um detector de participação e dado que f nós podem falhar, onde $f < \frac{k}{2} < n$ (usando assinaturas) ou $f < \frac{k}{3} < n$ (não usando assinaturas). O algoritmo DISCOVERY (algoritmo 2), executado por cada participante p do sistema, satisfaz as seguintes propriedades:*

- Terminação: *p termina a execução e retorna uma lista contendo nós conhecidos de p;*
- Exatidão: *o algoritmo DISCOVERY retorna o conjunto maximal de nós alcançáveis (conhecidos) por p em G_{di}.*

Prova: *Terminação.* O algoritmo termina quando p receber mensagens (respostas) de pelo menos todos os processos corretos alcançáveis (linha 32), como Π é finito e através das propriedades do protocolo de disseminação (seção 3.3), é garantido que esta condição acabará por ser satisfeita. *Exatidão.* O algoritmo apenas termina quando restarem no máximo f pendências, as quais podem estar divididas entre processos que informam vizinhos que não existem no sistema ($i.nei\_pend$) e processos dos quais p ainda não recebeu mensagens/respostas ($i.msg\_pend$). Além disso, na solução sem assinaturas (resp. com assinaturas) cada participante z (sendo z alcançável por p) do sistema é vizinho de pelo menos $3f+1$ (resp. $2f+1$) outros participantes, pois $f < \frac{k}{3} < n$ (resp. $f < \frac{k}{2} < n$). Desta forma, caso z seja malicioso, p computa mensagens (respostas) de no mínimo $2f + 1$ (resp. $f + 1$) vizinhos corretos de z, descobrindo z (linhas 16 e 22). Se z for correto, no pior caso p não computa mensagens de f vizinhos corretos de z ($i.msg\_pend$) e outros f vizinhos de z são faltosos, mas ainda assim, p computará $f + 1$ (resp. 1) mensagens de vizinhos corretos de z, descobrindo z (linhas 16 e 22). □

3.4.3. Determinação da Componente Poço

Esta fase visa estabelecer quais são os processos corretos que pertencem a componente poço do grafo de conhecimento induzido por um detector de participação *k*-OSR. Mais precisamente, através do processamento do algoritmo 3 (SINK), cada participante é capaz de determinar se é membro da componente poço deste grafo. A idéia de funcionamento deste algoritmo é que após a execução do procedimento DISCOVERY, os membros da componente poço obterão uma visão parcial do sistema que é necessariamente menor do que o conhecimento obtido pelos demais participantes, que conhecerão pelo menos os membros da componente a que pertencem e os membros da componente poço.

Na inicialização do algoritmo em um processo i, i executa o procedimento DISCOVERY com o intuito de estabelecer suas relações de conhecimento (linha 6) e envia uma mensagem com seu conjunto de processos conhecidos para todos os participantes alcançáveis/conhecidos (linha 7). Quando estas mensagens são entregues por algum participante, o mesmo responde para i com *ack* caso seu conjunto de conhecidos for igual ao de i (pertencem a mesma componente). Caso contrário, a resposta enviada para i é um *nack* (linhas 8-13).

No processamento das respostas em i (linha 14-26), i atualiza o conjunto de processos que responderam (linha 15). Além disso, caso a resposta recebida seja *nack*, o conjunto de processos que pertencem a outras componentes é atualizado (linha 17) e caso o número de processos que não pertencem a mesma componente de i seja maior do que f (linha 18), i conclui que não pertence à componente poço (linhas 18-20). No entanto, caso i receba respostas de todos os processos conhecidos, excluindo-se os possíveis faltosos (linha 23), e o número de processos que pertencem a outras componentes ($i.nacked$) não é maior do que f, i conclui que pertence à componente poço (linhas 24-25).

Lema 2 *Considerando um detector de participação k-OSR (ou equivalente) e dado que f nós podem falhar, onde $f < \frac{k}{2} < n$ (usando assinaturas) ou $f < \frac{k}{3} < n$ (não usando assinaturas). O algoritmo SINK (algoritmo 3), executado por cada participante p do sistema que possuí no mínimo $3f + 1$ nós na componente poço, satisfaz as seguintes propriedades:*

- Terminação: *p termina a execução determinando se pertence ou não à componente poço;*
- Exatidão: *p é membro da componente poço sse o algoritmo SINK retornar true.*

Algoritmo 3 Algoritmo SINK executado pelo participante i

```
constant:
(1) f:int //número máximo de faltas suportadas

variables:
(2) i.known:set of nodes //conjunto de processos conhecidos de i
(3) i.responded:set of nodes //conj.  de processos que se comunicaram com i
(4) i.nacked:set of nodes //conj.  de processos que ∉ compomente de i
(5) i.in_the_sink:boolean //variável indicando se i ∈ poço

** All Nodes **
INIT:
(6) i.known = DISCOVERY(); i.responded {i}; i.nacked = ∅;
(7) dependable_send(i.known,i);

(8) upon execution of dependable_deliver(sender.known,sender,routes)
(9)     if i.known == sender.known then
(10)      dependable_reply_send(ack,i,routes)
(11)    else
(12)      dependable_reply_send(nack,i,routes)
(13)    end if

(14) upon execution of dependable_reply_deliver(reply,sender)
(15)    i.responded = i.responded ∪ {sender}
(16)    if reply == nack then
(17)      i.nacked = i.nacked ∪ {sender};
(18)      if|i.nacked| ≥ f+1 then
(19)        i.in_the_sink = false;
(20)        return(i.in_the_sink,i.known);
(21)      end if
(22)    end if
(23)    if |i.responded| ≥ |i.known| - f  then
(24)      i.in_the_sink = true;
(25)      return(i.in_the_sink,i.known);
(26)    end if
```

Prova: *Terminação*. Em cada participante p, o algoritmo retorna quando p receber ou (*i.*) $f + 1$ respostas de processos de outras componentes (linha 18) ou (*ii.*) respostas de pelo menos todos os processos corretos determinados na inicialização (linha 23). Através das propriedades do protocolo de comunicação (seção 3.3), mesmo que $f < \frac{k}{2}$ ou $f < \frac{k}{3}$ (dependendo do uso de assinaturas) participantes falhem, é garantido que ou (*i.*) ou (*ii.*) sempre ocorrerá. *Exatidão*. Através do Lema 1, é garantido que ao final do procedimento DISCOVERY todos os processos corretos de uma mesma componente obtém o mesmo conhecimento. Assim, como os membros da componente poço apenas recebem respostas de membros da própria componente, é garantido que estes participantes concluam corretamente (linha 23). Além disso, os membros de uma componente não poço computarão no mínimo $f + 1$ mensagens de membros corretos da componente poço (que têm necessariamente um conhecimento menor sobre Π - Lema 1), pois adquiriram o conhecimento sobre pelo menos os $2f + 1$ participantes corretos da componente poço (Lema 1). Assim, através destas $f + 1$ mensagens os membros de componentes não poço concluirão corretamente (linha 18). □

3.4.4. Realização do Consenso

Esta é a etapa final na execução do consenso. A idéia principal é que os membros da componente poço executem um consenso bizantino clássico (ex. Paxos Bizantino [Castro and Liskov 2002]) e enviem o valor da decisão para os demais participantes do sistema. A resiliência ótima para algoritmos de consenso bizantino clássico é $3f + 1$ [Castro and Liskov 2002]. Sendo assim, é necessário a presença de no mínimo $3f + 1$ participantes na componente poço.

O algoritmo 4 (CONSENSUS) representa este processamento. Na inicialização, cada participante determina se pertence à componente poço e obtêm suas relações de conhecimento

(procedimento SINK - linha 9). Dependendo do resultado obtido, dois comportamentos distintos são possíveis: **(1)** Os membros da componente poço executam um consenso bizantino clássico (linha 11) e enviam a decisão para os demais participantes (linhas 17-19 e 21-26). Por construção, os membros corretos da componente poço estão presentes na visão de cada outro membro desta componente (Lema 1). Assim, como cada membro da componente poço conhece no mínimo $2f + 1$ participantes corretos (membros da mesma componente), o acordo do protocolo de consenso bizantino clássico sempre será obtido, juntamente com as outras propriedades definidas para estes protocolos de consenso [Castro and Liskov 2002]. Note que, as trocas de mensagens realizadas durante o estabelecimento deste consenso podem ser implementadas através do protocolo de comunicação definido na seção 3.3. **(2)** Os membros das outras componentes solicitam o valor da decisão aos participantes conhecidos, i.e., participantes alcançáveis, que inclui os membros do poço (linha 13). Um participante decidirá por um valor $value$ somente após receber $value$ de no mínimo $f + 1$ outros participantes, garantindo a presença de no mínimo um participante correto (linhas 27-34).

Algoritmo 4 Algoritmo CONSENSUS executado pelo participante i

```
constant:
(1) f:int //número máximo de faltas suportadas

input:
(2) i.initial:value //valor a ser proposto por i (input)

variables:
(3) i.in_the_sink:boolean //variável indicando se i ∈ poço
(4) i.known:set of nodes //conjunto de processos conhecidos de i
(5) i.decision:value //valor de decisão
(6) i.asked:set of ⟨node,routes⟩ tuples//conj.  de processos que solicitaram o valor de decisão
(7) i.values:set of ⟨node,value⟩ tuples //conjunto de decisões de outros nós

** All Nodes **
INIT: {Main Decision Task}
(8)    i.decision = ⊥; i.values = i.asked = ∅;
(9)    (i.in_the_sink,i.known) = SINK();
(10)   if i.in_the_sink then {Underline classical byzantine consensus with all p ∈ i.known}
(11)     Consensus.propose(i.initial);
(12)   else
(13)     dependable_send(GET_DECISION,i);
(14)   end if

** Node In Sink **
(15) upon Consensus.decide(v)
(16)    i.decision = v;
(17)    for each ⟨j,routes⟩ ∈ i.asked  do
(18)      dependable_reply_send(i.decision,i,routes);
(19)    end for
(20)    return(i.decision);

(21) upon execution of dependable_deliver(GET_DECISION,sender,routes)
(22)    if i.decision == ⊥ then
(23)       i.asked = i.asked ∪ {⟨sender,routes⟩};
(24)    else
(25)       dependable_reply_send(i.decision,i,routes);
(26)    end if

** Node Not In Sink **
(27) upon execution of dependable_reply_deliver(value,sender)
(28)    if i.decision == ⊥ then
(29)       i.values = i.values ∪ {⟨sender,value⟩};
(30)       if #⟨∗,value⟩ ∈ i.values ≥ f+1 then
(31)          i.decision = value;
(32)          return(i.decision);
(33)       end if
(34)    end if
```

Teorema 2 *O detector de participação k-OSR (ou equivalente) é suficiente para resolver o BFT-CUP, dado que f nós podem falhar em um sistema parcialmente síncrono que possuí no mínimo $3f + 1$ nós na componente poço, onde $f < \frac{k}{2} < n$ (usando assinaturas) ou $f < \frac{k}{3} < n$ (não usando assinaturas).*

Prova: Todos os participantes corretos da componente poço determinam esta condição (Lema 2) e participam do algoritmo de consenso bizantino clássico. Sendo assim, como esta componente possui no mínimo $2f + 1$ participantes corretos e o sistema é parcialmente síncrono, é garantido que as propriedades do consenso clássico serão atendidas. Desta forma, os membros

da componente poço obtém a decisão do consenso (linha 15), enviam esta decisão para os demais participantes (linha 17-19 e 21-26) e retornam o valor da decisão (linha 20) *terminado* o algoritmo. Como a componente poço possui no mínimo $2f + 1$ participantes corretos, que respondem às solicitações dos demais participantes[11] mesmo que $f < \frac{k}{2}$ ou $f < \frac{k}{3}$ (dependendo do uso de assinaturas) participantes falhem (seção 3.3), é garantido que os membros das outras componentes receberão $f + 1$ mensagens contendo o mesmo valor de decisão, podendo *terminar* decidindo por este valor (linhas 27-34). A *integridade* é garantida através do teste da linha 28, onde cada participante correto decide apenas caso ainda não tenha decidido. Além disso, um conluio entre os f possíveis participantes faltosos não será suficiente para fazer com que algum processo decida por valores incorretos, garantindo a propriedade de *acordo* do consenso. Note que, a propriedade de *validade* do consenso é conseguida através do protocolo de consenso clássico subjacente, i.e., o valor de decisão será o valor proposto por algum membro da componente poço. Já as propriedades de *acordo*, *terminação* e *integridade* são estabelecidas através do protocolo de consenso subjacente em conjunto com o algoritmo CONSENSUS. Esta prova de corretude do algoritmo CONSENSUS atesta que o detector de participação *k*-OSR é suficiente para resolver o BFT-CUP. □

4. Trabalhos Relacionados

A maioria dos trabalhos sobre consenso encontrados na literatura considera que o conjunto de processos do sistema é estático e previamente conhecido por todos os participantes do sistema. No entanto, nos últimos anos surgiram algumas pesquisas visando o estudo deste problema em ambientes dinâmicos, onde o número de participantes e suas identidades são, *a priori*, desconhecidos. Neste sentido, o primeiro trabalho a abordar o consenso em ambientes dinâmicos foi [Cavin et al. 2004], que resolve este problema para uma rede assíncrona onde os processos não podem falhar (CUP). Além disso, este trabalho define que o detector de participação OSR (redutível a único poço) é necessário e suficiente para resolver o CUP nestes ambientes. Em [Cavin et al. 2005] este modelo é estendido para tolerar faltas de parada nos participantes do sistema (FT-CUP), que funciona corretamente desde que as falhas obedeçam a um padrão de falhas, i.e., os processos podem falhar desde que o grafo de conhecimento continue pertencendo a classe OSR, sendo que as falhas devem ser detectadas por meio de detectores de faltas perfeitos [Chandra and Toueg 1996].

Um estudo comparando as condições de conectividade do grafo de conhecimento e as condições de sincronia do sistema é apresentado em [Greve and Tixeuil 2007], onde é estabelecido que o FT-CUP admite solução em redes assíncronas enriquecidas com o mais fraco detector de faltas capaz de resolver o consenso, desde que se aumente o grau de conhecimento entre os participantes (k-OSR). Nosso trabalho estende esse resultado mostrando que uma conectividade maior é requerida quando os processos podem falhar de forma bizantina (BFT-CUP) em um sistema parcialmente sincrono (sincronia mínima requerida para a realização do $\diamond S$ [Chandra and Toueg 1996]). A tabela 1 apresenta uma comparação entre as exigências de conectividade e sincronismo de cada solução para o consenso com participantes desconhecidos.

5. Conclusões

Este trabalho apresenta duas soluções para o consenso bizantino entre participantes desconhecidos (BFT-CUP), com e sem o uso de assinaturas digitais. Para cada solução, é provado quais são as condições necessárias e suficientes para resolver o BFT-CUP. A solução que utiliza assinaturas exige o uso de certificados emitidos por uma autoridade certificadora reconhecida pelos participantes, mas requer menor conectividade no grafo de conhecimento e menor número de envio de mensagens entre os participantes, resultando em menor custo de transmissão de dados.

[11] Note que todos os participantes do sistema alcançam os participantes da componente poço.

O modelo adotado neste trabalho, bem como os empregados nas soluções do FT-CUP [Cavin et al. 2005, Greve and Tixeuil 2007], suporta a mobilidade dos nós, mas não é forte o suficiente para tolerar entradas e saídas arbitrárias (as saídas podem ser computadas como faltas). Note que, após as relações de conhecimento serem estabelecidas, novos nós apenas serão considerados em execuções futuras do consenso.

Solução	modelo de faltas	detector de participação	k	participantes no poço/fonte	conectividade entre componentes	sincronismo
CUP [Cavin et al. 2004]	sem faltas	OSR	–	1	OSR	assíncrono
FT-CUP [Cavin et al. 2005]	paradas	OSR	–	1	OSR + padrão de falhas	assíncrono + P
FT-CUP [Greve and Tixeuil 2007]	paradas	k-OSR	$f+1$	$2f+1$	k caminhos disjuntos nos nós	assíncrono + $\diamondsuit S$
BFT-CUP com assinatura (este artigo)	bizantinas	k-OSR ou k-OFR	$2f+1$	$3f+1$	k caminhos disjuntos nos nós	parcialmente síncrono
BFT-CUP sem assinatura (este artigo)	bizantinas	k-OSR ou k-OFR	$3f+1$	$3f+1$	k caminhos disjuntos nos nós	parcialmente síncrono

Tabela 1. Características das soluções do consenso com participantes desconhecidos.

O principal resultado deste artigo é demonstrar que com o mesmo detector de participação e sincronia usada para resolver consenso com faltas por parada em sistemas auto-organizáveis podemos resolver este problema com faltas bizantinas, desde que se aumente a conectividade do grafo e o número de elementos em seu poço (ou fonte). Além disso, definimos um protocolo de disseminação tolerante a faltas bizantinas que é empregado em nossos protocolos e interessante por si só, podendo ser utilizado em outros protocolos para redes auto-organizáveis.

Referências

Awerbuch, B., Holmer, D., Nita-Rotaru, C., and Rubens, H. (2002). An on-demand secure routing protocol resilient to byzantine failures. In *WiSE '02: Proceedings of the 1st ACM workshop on Wireless security*, pages 21–30, New York, NY, USA. ACM.

Castro, M., Druschel, P., Ganesh, A., Rowstron, A., and Wallach, D. S. (2002). Secure routing for structured peer-to-peer overlay networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):299–314.

Castro, M. and Liskov, B. (2002). Practical Byzantine fault-tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461.

Cavin, D., Sasson, Y., and Schiper, A. (2004). Consensus with unknown participants or fundamental self-organization. In *Proc. of the 3rd Int. Conf. on Ad hoc Networks and Wireless*, pages 135–148.

Cavin, D., Sasson, Y., and Schiper, A. (2005). Reaching agreement with unknown participants in mobile self-organized networks in spite of process crashes. Technical Report IC/2005/026, EPFL - LSR.

Chandra, T. D. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267.

Dolev, D. (1982). The Byzantine generals strike again. *Journal of Algorithms*, (3):14–30.

Dwork, C., Lynch, N. A., and Stockmeyer, L. (1988). Consensus in the presence of partial synchrony. *Journal of ACM*, 35(2):288–322.

Fischer, M. J., Lynch, N. A., and Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382.

Greve, F. G. P. and Tixeuil, S. (2007). Knowledge connectivity vs. synchrony requirements for fault-tolerant agreement in unknown networks. In *Proc. of the Int. Conf. on Dependable Systems and Networks - DSN 2007*, pages 82–91.

Lamport, L., Shostak, R., and Pease, M. (1982). The Byzantine generals problem. *ACM Transactions on Programing Languages and Systems*, 4(3):382–401.

Avaliação de Replicação de Dados Estruturados Mutáveis em Sistemas Peer-to-Peer

Alexandre Nodari , Alcides Calsavara , Luiz Lima

[1]Programa de Pós-Graduação em Informática Aplicada
Pontifícia Universidade Católica do Paraná
Rua Imaculada Conceição, 1155, Prado Velho
80215-901 Curitiba, PR

{nodari,alcides,laplima}@ppgia.pucpr.br

Abstract. *A method to evaluate optimistic protocols for mutable data replication in peer-to-peer systems is proposed. Its use is shown through the simulation of four distinct replica protocols with respect to replica update and reconciliation in case a replica reenters the system. The results of the simulation permit to observe the influence on the performance of each protocol of the number of replicas, the chance of each node to reply messages sent to it and the rate of write operations. The carried out experiments permit to conclude that it is appropriate to employ optmistic data replication in peer-to-peer systems if some conditions are satisfied.*

Resumo. *É proposta uma metodologia para avaliação de protocolos de replicação otimista de dados mutáveis em sistemas peer-to-peer e é mostrada a sua aplicação através de experimentos de simulação com quatro protocolos de replicação que diferem entre si na maneira como fazem atualização das réplicas e a reconciliação em caso de retorno de uma réplica ao sistema. Os resultados da simulaçao permitem observar a influência do número de réplicas, da chance de cada nó responder às mensagens recebidas e da taxa de operações de escrita no desempenho de cada protocolo. O experimento realizado permite concluir que o emprego de replicação otimista de dados é viável em sistemas peer-to-peer se algumas condições forem observadas.*

1. Introdução

Os mecanismos de replicação atualmente empregados em sistemas peer-to-peer atuam primordialmente sobre dados não estruturados e imutáveis, pois sua principal utilização é nas aplicações de compartilhamento de arquivos multimídia, como discutem [Cuenca-Acuna et al., 2003] e [Lin et al., 2004], a fim de proporcionar maior disponibilidade e mais rapidez no acesso aos dados. Nessas aplicações, a replicação é implícita no mecanismo de compartilhamento, usando eventualmente fragmentos dos arquivos compartilhados. Como tanto os arquivos quanto seus fragmentos são dados imutáveis, não é necessário utilizar um mecanismo de replicação que mantenha consistência entre as réplicas. Além de imutáveis, os dados não são estruturados, o que simplifica a sua fragmentação. Porém, a evolução desses sistemas e o crescimento de outros tipos de aplicações, como as de comunicação e colaboração, ou até a convergência da arquitetura peer-to-peer com a de computação em grade sugere também

a necessidade de replicação de dados estruturados e mutáveis, como discutido em [Androutsellis-Theotokis and Spinellis, 2004].

De fato, as oportunidades para o emprego de replicação de dados estruturados mutáveis são muitas, tanto para fins de melhoria no desempenho como para fins de tolerância a faltas. Algumas aplicações de compartilhamento de arquivos replicam metadados sobre esses arquivos para permitir consultas mesmo em nós desconectados e ainda permitir consultas estruturadas como, por exemplo, buscar músicas de um certo intérprete ou de um certo álbum. Um sistema de arquivos distribuído tira proveito da replicação de metadados para aperfeiçoar o acesso aos arquivos, como em [Bolosky et al., 2000]. Da mema forma, uma ferramenta de armazenamento distribuída pode aperfeiçoar o acesso aos dados, como descrito em [Kubiatowicz et al., 2000]. Aplicações de comunicação e colaboração poderiam evitar a necessidade de um servidor central se as funcionalidades de autenticação, inicialização e atualização de lista de contatos estivessem replicadas. Aplicações distribuídas poderiam replicar dados de controle de redes, como descrito em [Granville et al., 2005], ou replicar dados de execução da aplicação em grade de forma a se tornarem adaptáveis ao ambiente, redirecionando tarefas para nós com maior banda ou maior processamento disponível, conforme [Schuler et al., 2003].

Embora a teoria de replicação de dados seja bem consolidada, a implementação de protocolos de replicação em sistemas peer-to-peer exige um projeto adequado às suas características mais acentuadas, tais como a alta latência e a entrada e saída constantes de nós. O nível de consistência de dados mutáveis armazenados em uma DHT é verificado experimentalmente em [Picconi et al., 2007]. A DHT é hospedada em uma rede peer-to-peer, na qual os nós estão sujeitos a variados níveis de entrada e saída e a consistência é mantida usando-se o algoritmo baseado em quorum de operações de leitura e escrita. Observa-se que o algoritmo é muito sensível aos eventos de entrada e saída de nós, pois isso dificulta bastante a obtenção de quorum. Em [Antoniu et al., 2004], um protocolo hierárquico de consistência de réplicas é proposto e avaliado no contexto de computação em grade sobre redes peer-to-peer. O principal objetivo do protocolo é separar claramente a manutenção da réplicas do gerenciamento de tolerância a faltas. A avaliação feita para esse protocolo em particular permitiu verificar o seu desempenho quanto a latência de operações de leitura e escrita nos dados.

Neste artigo, propomos uma metodologia para avaliação de protocolos de replicação otimista de dados mutáveis em sistemas peer-to-peer e mostramos a sua aplicação através de experimentos de simulação com quatro protocolos de replicação. Todos esses protocolos são *single-master*, mas diferem na maneira como fazem a atualização das réplicas e a reconciliação em caso de retorno de uma réplica ao sistema. Cada protocolo é avaliado em simulações nas quais variam-se o número de réplicas de cada master, a chance de cada nó responder às mensagens recebidas e a taxa de operações de escrita.

O restante deste artigo está organizado como segue. A Seção 2 discute os principais conceitos de replicação de dados que influenciaram na metodologia proposta e no experimento realizado. A Seção 3 comenta sobre plataformas para a construção de sistemas peer-to-peer e justifica a escolha de uma plataforma específica para a realização do experimento de avaliação de protocolos de replicação. A Seção 4 apresenta a metodologia proposta e descreve as condições para a realização do experimento da metodologia. A Seção 5 descreve a aplicação utilizada no experimento. A Seção 6 apresenta os principais

resultados obtidos no experimento. A Seção 7 discute esses resultados. Finalmente, a Seção 8 apresenta as principais conclusões sobre o trabalho realizado e comenta sobre possíveis trabalhos futuros.

2. Replicação de Dados Mutáveis

Para protocolos de replicação em sistemas distribuídos que tratam de informações mutáveis, ou seja, aceitam operações de escrita, existem duas classificações. A primeira determina qual(is) nó(s) recebe(m) as operações de escrita e como e quando essas operações são transferidas para as réplicas [Lung et al., 2004], determinando as opções *passiva* e *ativa*. Outra forma de classificação de protocolos de replicação em sistemas distribuídos determina como lidar com operações de escrita e leitura concorrentes, determinando as opções *pessimista* e *otimista*. Na replicação otimista não há a necessidade de sincronização durante o acesso as réplicas: permite que a leitura e escrita de dados sejam feitas assumindo de forma "otimista" que conflitos raramente irão acontecer e, se acontecerem, podem ser resolvidos.

A replicação pessimista funciona bem em redes locais, onde a latência de rede é pequena e falhas não são comuns. Porém, a Internet, que é onde normalmente se configuram os sistemas peer-to-peer, ainda é lenta e pouco confiável. Além disso, redes formadas por computadores móveis, com conectividade intermitente, estão se tornando mais populares. Protocolos pessimistas também não escalam bem em ambientes distribuídos. Por outro lado, a replicação otimista de dados aumenta a sua disponibilidade, possibilitando melhora no desempenho das aplicações. Aumenta-se também a escalabilidade do número de nós participantes ao relaxar a sincronização necessária entre as réplicas. Um bom equilíbrio entre velocidade de propagação de operações de escrita, número de réplicas e consumo de banda é importante. Quanto mais rapidamente as alterações forem propagadas para as réplicas, maior é a convergência entre as réplicas e, ainda, menor é a probabilidade de um conflito ocorrer. Em sistemas peer-to-peer o fator preocupante fica com a taxa normalmente elevada de entrada e saída de nós da rede. Isso pode fazer com que muitas mensagens enviadas no sistema não cheguem ao seu destino.

Uma questão crítica no projeto de replicação otimista refere-se ao número de réplicas que podem aceitar operação de escrita. Os sistemas que só possuem uma réplica que aceita operações de escrita são denominados *single-master*, enquanto que sistemas onde mais de uma réplica aceita operações de escrita são denominados *multi-master*. Sistemas single-master são mais simples, porém têm sua disponibilidade para operações de escrita reduzida, principalmente em sistemas com operações de escrita freqüentes. A escalabilidade do sistema single-master está limitada ao desempenho da única instância que recebe operações de escrita. Sistemas multi-master oferecem maior disponibilidade, mas com um aumento significativo de complexidade. E apesar da escalabilidade ser maior que em um sistema single-master, ao aumentar o número de réplicas que aceitam operações de escrita aumenta-se também a taxa de conflitos entre operações, determinando que esta vantagem não seja normalmente muito grande.

Existem duas maneiras de transmitir alterações em dados para suas réplicas. A primeira é transferir todas as informações novamente, sobrepondo as informações existentes. Esse mecanismo é denominado transferência de estado. Outra forma é enviar somente a operação que causou a alteração, denominada transferência de operações. Para

definir quando transferir uma operação ou estado pode-se optar pela técnica *push*. Nesta técnica o nó que recebe uma operação inicia imediatamente a propagá-la para os outros nós. Isso é recomendado, pois ajuda a diminuir a divergência entre réplicas e evita buscas (*pull*) desnecessárias. Técnicas *pull* não são recomendadas para redes dinâmicas por aumentar o número de mensagens sobre um ambiente dinâmico e mais propenso a falhas. Mas em sistemas single-master é possível usar uma técnica híbrida na qual usualmente a réplica master propaga (push) as novas operações, mas também uma réplica que está se conectando a rede pode buscar (pull) na réplica master as últimas alterações.

3. Infra-estrutura para Sistemas Peer-to-Peer

Sistemas peer-to-peer têm sido bastante pesquisados devido ao sucesso de algumas aplicações comerciais. Com isso surgiram muitas opções de infra-estrutura para a construção desses sistemas. Entre essas, destacam-se os arcabouços JXTA e Pastry, pois têm recebido muita atenção nas pesquisas recentes, além de oferecem implementações públicas e gratuitas.

JXTA [1][JXTA, 2008] Uma especificação de arcabouço para sistemas peer-to-peer independente de linguagem de programação, plataforma para comunicação entre dispositivos, localização física e tecnologia de rede no qual se encontram instalados. Para tanto, mapeia os principais conceitos de computação distribuída numa série de entidades e elementos de comunicação, com distinção e hierarquia entre os peers de acordo com suas funções específicas. Basicamente, peers transmitem mensagens apenas através de *pipes*, isto é, canais virtuais que são, em geral, unidirecionais e não-confiáveis, e que se conectam a um ponto de entrada e outro de saída (*end points*). Mensagens são documentos XML que carregam em seu cabeçalho, além do identificador da fonte, a informação de roteamento necessária, tal como a seqüência de peers a ser percorrida.

Pastry [Rowstron and Druschel, 2001] Um arcabouço para a construção de redes peer-to-peer sobrepostas à Internet, com mecanismos para localização de recursos e roteamento de mensagens. Os nós de uma rede Pastry são organizados em forma de anel, sendo que cada nó recebe um identificador único de 128 bits, isto é, a quantidade de nós na rede peer-to-peer pode variar entre 0 e 2^{128}. Há ainda mecanismos no arcabouço que permitem que a rede se recomponha automaticamente em caso de falhas, tornando-a relativamente robusta. O roteamento de mensagens baseia-se simplesmente numa função logarítmica que é aplicada a cada passo do roteamento.

Optamos pelo Pastry para a realização do experimento pelas seguintes razões:

1. Escalabilidade é um dos principais requisitos de sistemas peer-to-peer. O fato de JXTA utilizar o conceito de peers diferenciados para certas funções contribui negativamente para esse fim. Por outro lado, Pastry trabalha com redes completamente descentralizadas, que é uma uma característica que aumenta o potencial de escalabilidade.
2. Simplicidade é uma característica bastante desejável para o arcabouço, pois facilita o desenvolvimento de protótipos. Nesse ponto, a arquitetura e a interface de programação do Pastry são vantajosas em relação ao JXTA.

[1] Juxtapose

3. Facilidade de simulação é fundamental na realização de experimentos envolvendo computação distribuída. O arcabouço Pastry possui uma interface de programação específica para esse fim através da qual é possível simular uma rede peer-to-peer, sem perda de generalidade. Isto é, a plataforma de simulação para peer-to-peer permite avaliar aplicações com o mesmo grau de fidelidade que se teria em um sistema peer-to-peer real.

4. Metodologia e Ambiente de Avaliação

4.1. Nomenclatura

Os seguintes termos são definidos como parte da metodologia proposta para avaliação de protocolos de replicação otimista em sistemas peer-to-peer:

comando Uma mensagem contendo uma operação de escrita.

intensidade de escrita Chance de um nó enviar um comando ao invés de enviar uma mensagem comum (contendo uma consulta ou uma resposta). Por exemplo, se a intensidade de escrita é de 10%, cada nó envia aproximadamente 10 comandos e 90 mensagens comuns a cada 100 mensagens.

chance de vida Chance de um peer estar operante, ou seja, receber e, se necessário, responder as mensagens que lhe sejam enviadas. Por exemplo, se a chance de vida for 10%, cada nó recebe aproximadamente 10 em cada 100 mensagens enviadas para ele.

réplica efetiva Réplica que recebe pelo menos um comando executado no master. Isso permite ao nó que contém a réplica estar ciente que dados de um outro nó estão replicados nele. Assim é possível ao nó responder consultas sobre esses dados e executar reconciliações com o nó master desses dados.

4.2. Protocolos de Replicação

Somente os protocolos single-master de replicação foram avaliados. Futuramente, outras combinações de estratégia de propagação de atualização e de conciliação deverão ser experimentadas e avaliadas.

StatePush–NullSync A cada comando executado em um master, uma mensagem de atualização de estado é criada e enviada (em multicast) para o grupo de réplicas. Assim, na atualização das réplicas, há duas latências e o número de mensagens é da $O(n)$, onde n é o número de réplicas. Não ocorre reconciliação quando um nó volta a operar.

CommandPush–NullSync A cada execução de comando em um master, o próprio comando é enviado (em multicast) para execução em cada réplica. Assim, na atualização das réplicas, há duas latências e o número de mensagens é da $O(n)$, onde n é o número de réplicas. Não ocorre reconciliação quando um nó volta a operar.

CommandPush–StatePullSync A cada execução de um comando em um master, o próprio comando é enviado para execução em cada réplica. Assim, na atualização das réplicas, há duas latências e o número de mensagens é da $O(n)$, onde n é o número de réplicas. Ocorre reconciliação baseada em *estado* toda a vez que um nó contendo réplicas volta a operar: para cada réplica que o nó mantém, consulta nos respectivos masters os seus estados atuais. Assim, na reconciliação, há duas

latências e o número de mensagens é da $O(n)$, onde n é o número de masters consultados.

CommandPush–CommandPullSync A cada execução de um comando em um master, o próprio comando é enviado para execução em cada réplica. Assim, na atualização das réplicas, há duas latências e o número de mensagens é da $O(n)$, onde n é o número de réplicas. Ocorre reconciliação baseada em *comando* toda a vez que um nó contendo réplicas volta a operar: para cada réplica que o nó mantém, consulta nos respectivos masters os correspondentes comandos que deixou de executar. Assim, na reconciliação, há duas latências e o número de mensagens é da $O(n)$, onde n é o número de masters consultados.

4.3. Métricas de Avaliação

As seguintes medições são realizadas para a avaliação dos protocolos de replicação:

1. Número de mensagens trocadas entre os nós: permite verificar o *overhead* de comunicação provocado pelo protocolo.
2. Tamanho das mensagens em relação ao tamanho dos dados: permite verificar o consumo de banda provocado pelo protocolo.
3. Taxa de réplicas efetivas (medida através do percentual de réplicas efetivas em relação ao número de réplicas configurado para a simulação): permite verificar o nível de participação das réplicas.
4. Qualidade das réplicas efetivas (medida através da média dos percentuais de comandos aplicados corretamente em cada réplica em relação ao total de comandos aplicados no master): permite verificar a própria qualidade das réplicas, isto é, o nível de correção das réplicas.

4.4. Ambiente de Experimentação

O experimento foi realizado através da simulação de uma rede peer-to-peer de 100 nós, usando o arcabouço Pastry em um computador com processador Athlon 2600 e 1GB de memória e o desenvolvimento da aplicação alvo na linguagem Java, com as seguintes condições:

1. Cada nó da rede é *master* de seus próprios dados, enquanto outros nós armazenam réplicas desses dados. Assim, todos os nós são *master* e ainda podem armazenar réplicas de dados de outros nós. O número de réplicas esperadas é único em cada simulação, isto é, cada *master* tem os seus dados replicados o mesmo número de vezes, sendo que a alocação de réplicas é feita de maneira aleatória. Por exemplo, se o número de réplicas é configurado em quatro, os dados de cada master são replicados em outros quatro nós escolhidos aleatoriamente no conjunto completo de nós.
2. Em cada simulação são utilizados os 100 nós da rede, sendo que cada um envia 100 mensagens. Na prática, 100 é um número relativamente baixo para redes peer-to-peer reais, mas o equipamento disponível para o experimento impôs tal restrição, deixando experimentos com redes maiores para trabalhos futuros.
3. A cada mensagem enviada, um tipo de mensagem (comando ou mensagem comum) é escolhido de acordo com o parâmetro *intensidade de escrita*.
4. O recebimento das mensagens pelos nós depende da *chance de vida* especificada para a simulação.

5. As medições de taxa e qualidade de réplicas efetivas são feitas imediatamente após os ciclos de envio de mensagens a fim de evitar a convergência das réplicas nos protocolos de atualização com reconciliação, permitindo assim uma comparação mais justa entre os protocolos.
6. São feitas as seguintes variações nos parâmetros de simulação:
   (a) Número de réplicas esperadas: 1, 2, 4 e 8.
   (b) Chance de vida: 10%, 20%, 40% e 80%.
   (c) Intensidade de escrita: 10%, 20%, 40% e 80%.

5. Aplicação Alvo

Uma aplicação simples e, ao mesmo tempo, adequada para a verificação da metodologia de avaliação de protocolos de replicação proposta neste trabalho é o gerenciamento de listas de contatos pessoais. Atualmente a maioria dos sistemas existentes utiliza um servidor central que autentica os usuários e mantém suas listas de contatos. Esta lista de contatos é um exemplo de metadados que poderiam estar replicados, evitando a necessidade de um servidor central.

Como somente o próprio usuário pode alterar sua lista de contatos, o mecanismo de replicação pode ser *single-master*. O nó onde o usuário está conectado no momento é o *master*. Exemplos de comandos incluem inserção de um novo contato, remoção de um contato e edição de um contato. Se, por exemplo, o protocolo de replicação utilizado for o *CommandoPush-CommandPullSync*, ao receber um comando de escrita, o nó *master* propaga esse comando para todas as réplicas. As que estiverem operantes executam imediatamente o mesmo comando. Se alguma réplica não estiver operante, ao voltar a esse estado, obtém os comandos faltantes no *master*. Na ocorrência de comandos não comutativos, como por exemplo, uma inserção e uma deleção, a ordenação é feita no nó *master*. No caso de uma réplica receber comandos fora de ordem, ela deve ordená-los antes de executar ou obter novamente os comandos no *master*.

6. Resultados

Os resultados da simulação estão parcialmente representados pelos gráficos das Figuras 1 a 4. Os objetivos da exibição desses gráficos são o de auxiliar na discussão de certos fenômenos observados nos experimentos e o de mostrar o potencial de análise da metodologia de avaliação apresentada neste trabalho.

Os gráficos das Figura 1 a 3 referem-se ao protocolo de atualização de réplicas que denominamos *StatePush–NullSync*, definido na Seção 4.2. Em cada uma dessas Figuras, uma métrica de avaliação em particular (das quatro métricas definidas na Seção 4.3) é objeto dos correspondentes gráficos. A Figura 1 permite verificar o comportamento do número de mensagens em diversas situações. Observa-se que o número de mensagens cresce linearmente de acordo com o número de réplicas[2] também com a intensidade de escrita, mas há um certa independência quanto à chance de vida. A Figura 2 permite verificar a variação do tamanho médio das mensagens. Nota-se uma diferença significativa no tamanho médio das mensagens. Este aumenta com a intensidade de escrita, pois mais comandos são enviados e o tamanho dos comandos é usualmente maior que

[2]Para fins de legibilidade, foi feita interpolação nos gráficos em que o eixo horizontal representa o número de réplicas embora o seu domínio seja discreto.

o tamanho das mensagens comuns nesse protocolo; uma mensagem pode atingir até 40 vezes o tamanho dos dados. A Figura 3 permite verificar a qualidade das réplicas efetivas. Observa-se, principalmente, que a qualidade das réplicas efetivas aumenta com a chance de vida dos nós e com a intensidade de escrita da aplicação, pois assim aumentam-se as chances de uma réplica receber um comando enviado por seu master. O custo desse alto índice médio de qualidade é o tamanho elevado das mensagens, conforme já discutido. Essa mesma avaliação detalhada do protocolo *StatePush–NullSync* poderia ser feita para os três outros protocolos de replicação, mas aqui não dispomos do espaço necessário para isso.

Finalmente, a Figura 4 exemplifica uma simples comparação entre os quatro protocolos definidos, com relação às quatro métricas experimentadas. Tal simplicidade na comparação foi possível através da fixação de dois parâmetros de simulação, a saber o número de réplicas e a intensidade de escrita. Entretanto, a variação no parâmetro de chance de vida dos nós é adequada para a análise, pois tende a ser o parâmetro mais significativo em sistemas peer-to-peer. Nesse cenário em particular, analisando-se o tamanho médio das mensagens, percebe-se que os protocolos mais complexos, isto é, que fazem reconciliação, acabam por compensar o aumento no número de mensagens, pois só utilizam mensagens maiores durante as reconciliações. Os protocolos que fazem reconciliação dobram o número de mensagens no pior caso, mas o protocolo *StatePush–NullSync* gera mensagens que são mais que quatro vezes maior que o tamanho dos dados do nó. A razão entre réplicas efetivas e esperadas aumenta de acordo com a chance de um nó estar operante, conforme esperado. Mas é interessante ressaltar que, a partir de 40% de chance de vida, essa razão já é quase 100%. A qualidade das réplicas efetivas nos protocolos com reconciliação é intermediária em relação às outras duas simulações: esses protocolos se comportam melhor que o protocolo de transferência de operações e pior que o protocolo de transferência de estados para chances de vida inferiores a 40%. Para chances de vida superiores a 40% os protocolos com com reconciliação se comportam de maneira semelhante ao protocolo de transferência por estados.

7. Discussão

Observamos que a replicação otimista se torna bastante interessante quando a chance de vida dos nós é superior a 40%, ou seja, que os nós participantes da aplicação tenham uma taxa de entrada e saída da rede que possibilite que pelo menos 40% das mensagens cheguem com sucesso ao seu destino. Apesar de essa condição ser facilmente obtida em aplicações departamentais, algumas aplicações distribuídas mundialmente não apresentam essa característica. A partir dessa faixa de chance de vida, o número de mensagens é cada vez menor devido ao menor número de reconciliações necessárias. A quantidade de réplicas efetivas é alto mesmo em sistemas com intensidade de escrita baixa. A qualidade das réplicas é boa, partindo de aproximadamente 85% de convergência com o master e tendendo a 100% com o aumento da chance de vida.

O protocolo baseado em transferência de estados é bastante simples e oferece bons resultados mesmo com chance de vida inferior a 40%. Porém, apresenta uma desvantagem, pois aumenta o tamanho médio das mensagens. Isso pode ser um problema caso o tamanho dos dados em cada nó seja grande, ou crescer rapidamente com a execução de comandos de escrita. Esses problemas ainda podem ser acentuados caso o sistema tenha uma alta intensidade de escrita.

Os experimentos confirmaram que a técnica de transferência *push* é interessante para sistemas com grande quantidade de operações de escrita, pois melhora sensivelmente a convergência das réplicas com o master.

Os protocolos que usam reconciliação (por estado ou por comandos) diminuem consideravelmente a média de tamanho das mensagens. Em contrapartida, aumentam o número de mensagens. Porém, principalmente em sistemas com alta intensidade de escrita, o consumo total de banda é menor nessas configurações. Os dois protocolos com reconciliação apresentam resultados bastante parecidos, exceto pela diferença a favor da reconciliação por comandos no tamanho médio das mensagens. Esse ganho, entretanto, vem com um aumento da complexidade, pois nesse protocolo é necessário que cada nó guarde um histórico dos últimos comandos executados.

8. Conclusão

O trabalho realizado permite concluir que o emprego de replicação otimista de dados é viável em sistemas peer-to-peer se algumas condições forem observadas. A metodologia de avaliação proposta e o correspondente experimento realizado geraram resultados que mostram que mesmo protocolos simples, como o *push* com transferência de estados, pode oferecer bons resultados e que, aumentando a complexidade do protocolo, é possível obter um mecanismo de replicação com uma relação consumo de banda versus convergência das réplicas em relação ao master bastante interessante, como nos protocolos com reconciliação. Para muitos sistemas não é necessário um projeto de replicação otimista muito complexo. Portanto, o emprego de replicação otimista de forma bem controlada permite aperfeiçoar sistemas peer-to-peer e até mesmo possibilita funcionalidades que seriam inviáveis sobre um mecanismo de replicação pessimista.

Futuramente, seria interessante avaliar as respostas dos protocolos estudados para aplicações de larga escala. Se aumentado o número de nós nas simulações é possível efetuar tal avaliação. Outra alteração interessante nos parâmetros das simulações seria utilizar intensidades de escrita pequenas, bastante comuns em aplicações reais, como por exemplo, valores menores que 10%. Pode-se ainda evoluir a simulação de entrada e saída dos nós da rede, tornando o comportamento da simulação mais próximo da realidade de algumas aplicações atuais, incluindo a necessidade de lidar com a saída permanente de alguns nós. Outros protocolos de replicação de dados estruturados podem ser interessantes para outros tipos de informações de sistemas peer-to-peer. Replicação multi-master, com opções de detecção e resolução de conflitos [Martins et al., 2006] são um tema extenso de estudo. Finalmente, latência poderia ser verificada como uma métrica de avaliação dos protocolos.

Referências

Androutsellis-Theotokis, S. and Spinellis, D. (2004). A survey of peer-to-peer content distribution technologies. *ACM Comput. Surv.*, 36(4):335–371.

Antoniu, G., Deverge, J.-F., and Monnet, S. (2004). Building fault-tolerant consistency protocols for an adaptive grid data-sharing service. Technical Report 5309, IRISA, Rennes, França.

Bolosky, W. J., Douceur, J. R., Ely, D., and Theimer, M. (2000). Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *SIGMETRICS '00: Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 34–43, New York, NY, USA. ACM.

Cuenca-Acuna, F. M., Martin, R. P., and Nguyen, T. D. (2003). Autonomous replication for high availability in unstructured p2p systems. In *SRDS*, pages 99–108. IEEE Computer Society.

Granville, L.Z.and da Rosa, D., Panisson, A., Melchiors, C., Almeida, M., and Tarouco, L. (2005). Managing computer networks using peer-to-peer technologies. *Communications Magazine, IEEE*, 43(10):62–68.

JXTA (2008). JXTA Project. URL: http://www.jxta.org.

Kubiatowicz, J., Bindel, D., Chen, Y., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., and Zhao, B. (2000). Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM.

Lin, W. K., Chiu, D. M., and Lee, Y. B. (2004). Erasure code replication revisited. In *P2P '04: Proceedings of the Fourth International Conference on Peer-to-Peer Computing*, pages 90–97, Washington, DC, USA. IEEE Computer Society.

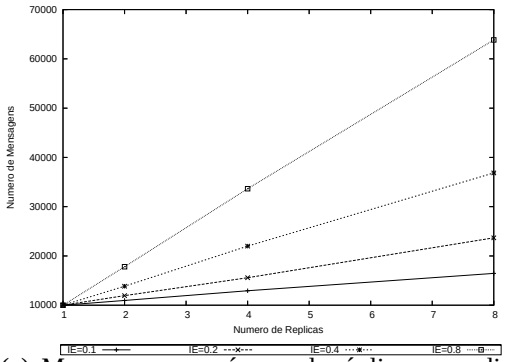
Lung, L., Bessani, A., and Fraga, J. (2004). Programação de sistemas distribuídos confiáveis. In *ERI-SC'04 – XII Escola Regional de Informática*, pages 1–40. SBC.

Martins, V., Akbarinia, R., Pacitti, E., and Valduriez, P. (2006). Reconciliation in the appa p2p system. In *ICPADS '06: Proceedings of the 12th International Conference on Parallel and Distributed Systems*, pages 401–410, Washington, DC, USA. IEEE Computer Society.

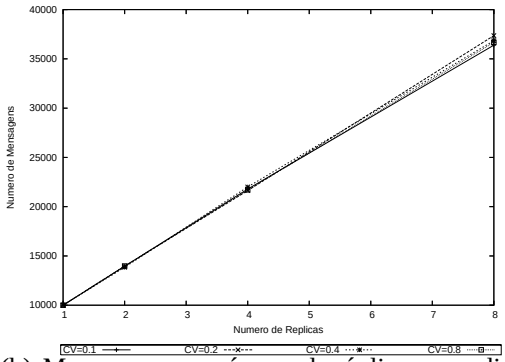
Picconi, F., Busca, J.-M., and Sens, P. (2007). An experimental evaluation of the Pastis peer-to-peer file system under churn. Technical Report 6114, INRIA, França.

Rowstron, A. and Druschel, P. (2001). Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350.

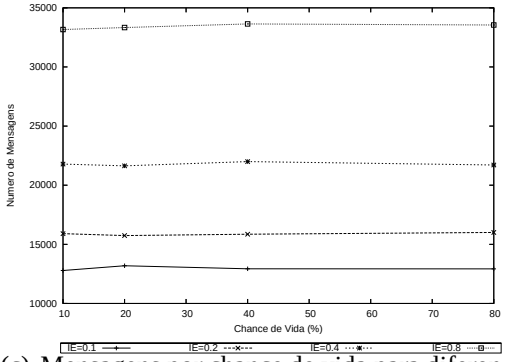
Schuler, C., Weber, R., Schuldt, H., and Schek, H. (2003). Peer-to-peer process execution with osiris. In *Proc. of First International Conference on Service-Oriented Computing ICSOC*.



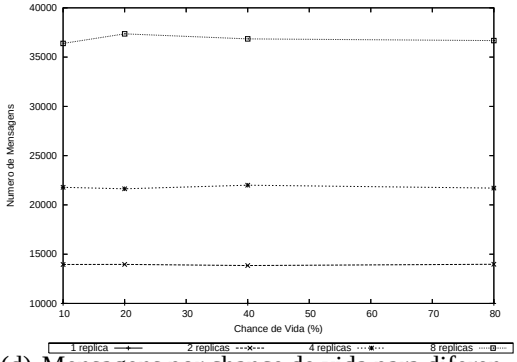
(a) Mensagens por número de réplicas para diferentes intensidades de escrita (IE) e chance de vida de 40%



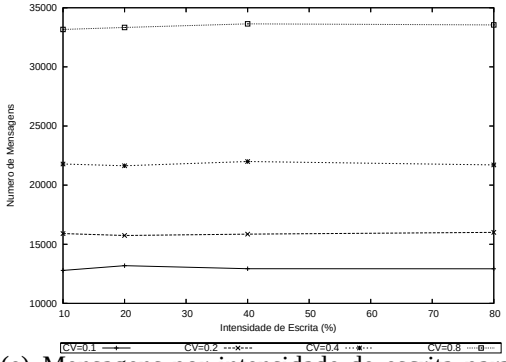
(b) Mensagens por número de réplicas para diferentes chances de vida (CV) e intensidade de escrita de 40%



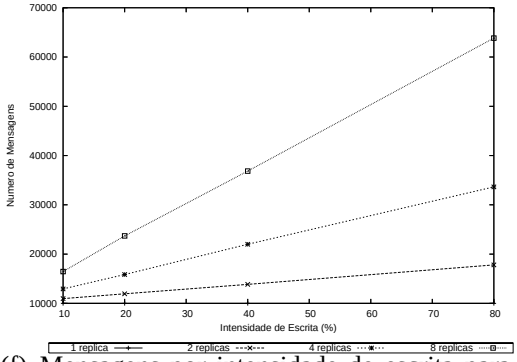
(c) Mensagens por chance de vida para diferentes intensidades de escrita (IE) e 4 réplicas



(d) Mensagens por chance de vida para diferentes números de réplicas e intensidade de escrita de 40%

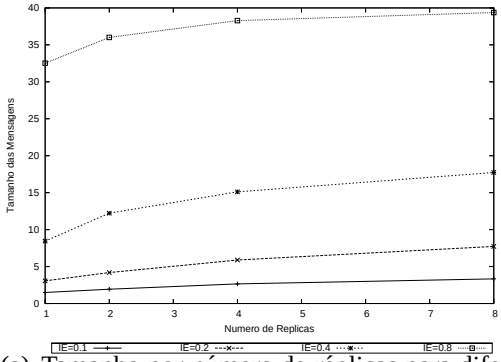


(e) Mensagens por intensidade de escrita para diferentes chances de vida (CV) e 4 réplicas

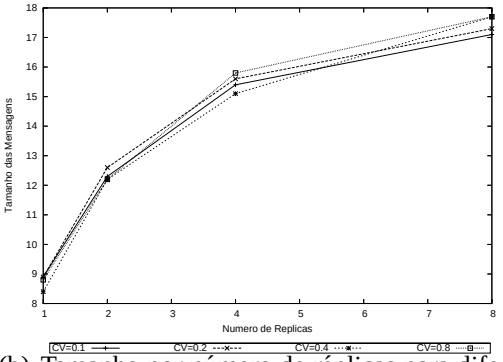


(f) Mensagens por intensidade de escrita para diferentes números de réplicas e chance de vida de 40%

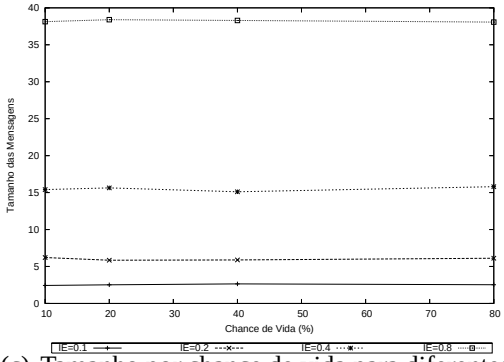
Figura 1. Número de mensagens no protocolo StatePush–NullSync



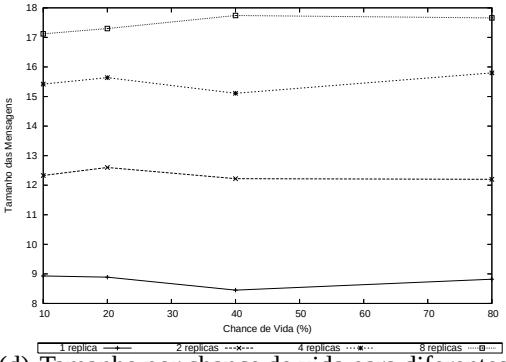
(a) Tamanho por número de réplicas para diferentes intensidades de escrita (IE) e chance de vida de 40%



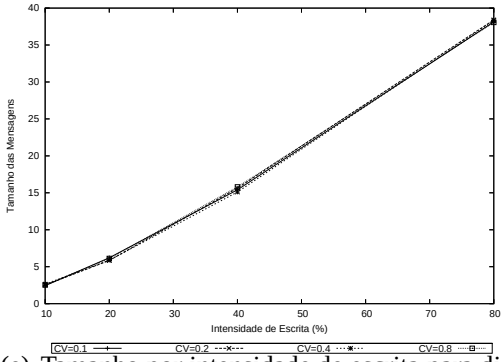
(b) Tamanho por número de réplicas para diferentes chances de vida (CV) e intensidade de escrita de 40%



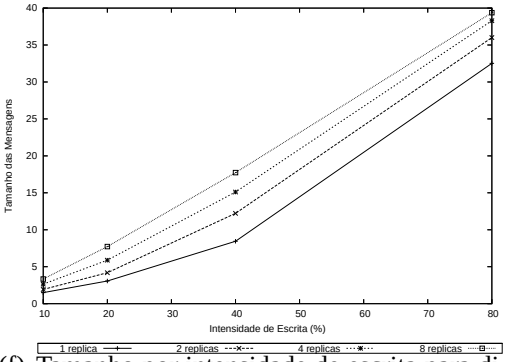
(c) Tamanho por chance de vida para diferentes intensidades de escrita (IE) e 4 réplicas



(d) Tamanho por chance de vida para diferentes números de réplicas e intensidade de escrita de 40%

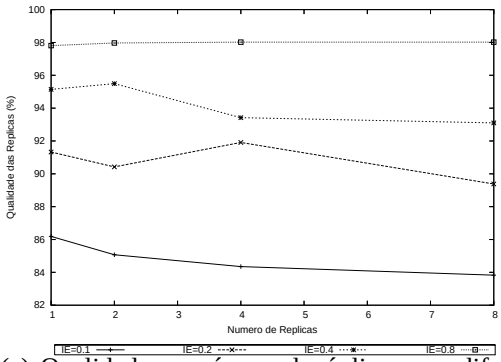


(e) Tamanho por intensidade de escrita para diferentes chances de vida (CV) e 4 réplicas

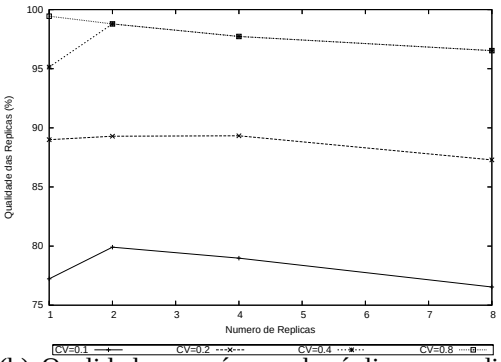


(f) Tamanho por intensidade de escrita para diferentes números de réplicas e chance de vida de 40%

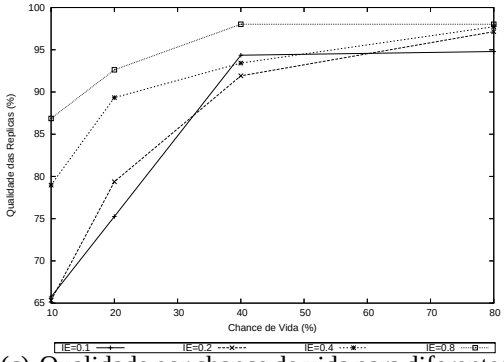
Figura 2. Tamanho das mensagens no protocolo StatePush–NullSync



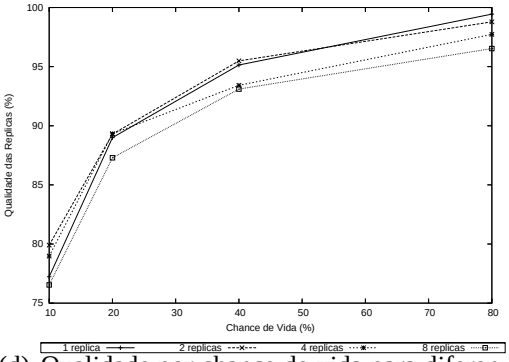
(a) Qualidade por número de réplicas para diferentes intensidades de escrita (IE) e chance de vida de 40%



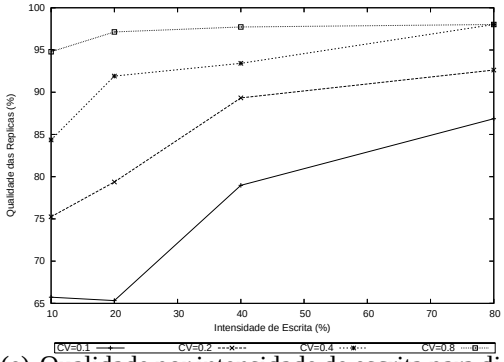
(b) Qualidade por número de réplicas para diferentes chances de vida (CV) e intensidade de escrita de 40%



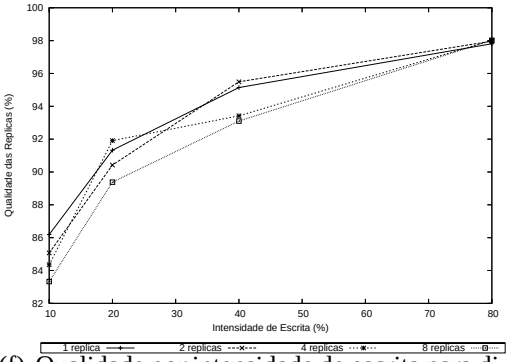
(c) Qualidade por chance de vida para diferentes intensidades de escrita (IE) e 4 réplicas



(d) Qualidade por chance de vida para diferentes números de réplicas e intensidade de escrita de 40%

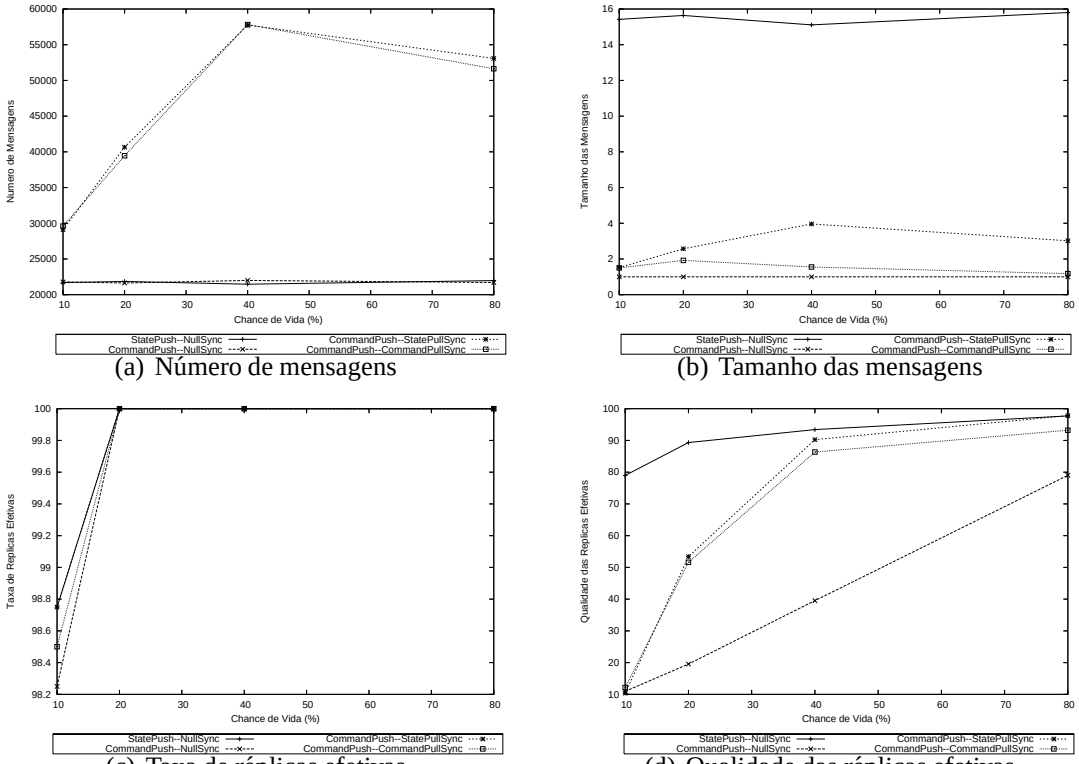


(e) Qualidade por intensidade de escrita para diferentes chances de vida (CV) e 4 réplicas



(f) Qualidade por intensidade de escrita para diferentes números de réplicas e chance de vida de 40%

Figura 3. Qualidade das réplicas efetivas no protocolo StatePush–NullSync



(a) Número de mensagens (b) Tamanho das mensagens

(c) Taxa de réplicas efetivas (d) Qualidade das réplicas efetivas

Figura 4. Comparação dos quatro protocolos por chance de vida para quatro réplicas e intensidade de escrita de 40%

Reconfiguração Dinâmica de Componentes em Sistemas Distribuídos de Controle e Supervisão, com Aplicação a Tolerância a Falhas

Neima Prado, Raimundo José de Araújo Macêdo, Luciano Porto Barreto

Laboratório de Sistemas Distribuídos (LaSiD)
Programa de Pós-Graduação em Mecatrônica
Departamento de Ciência da Computação, Universidade Federal da Bahia
Campus de Ondina, CEP: 40170-110, Salvador-BA, Brasil

npsantos@ufba.br, macedo@ufba.br, lportoba@ufba.br

Abstract. *The property of dependability for real-time distributed systems is closely related to its capability to adapt to environmental changes, especially for replacing faulty components. This paper presents the design and implementation of a service that supports dynamic reconfiguration of real-time component-based distributed systems in the control and supervision domain. The service described here was evaluated by the development of a cruise control application in which a faulty controller is replaced by a replica at runtime. The reconfiguration impact on the system performance was analysed by end-to-end control quality metrics and it was considered appropriate to the application being evaluated.*

Resumo. *A qualidade de confiança no funcionamento (dependability) de sistemas de tempo real distribuídos está intimamente ligada à sua capacidade de adaptação dinâmica, para se adequar às mudanças no ambiente, visando, em particular, a reposição de componentes defeituosos. O presente artigo descreve o projeto e implementação de um serviço de reconfiguração dinâmica para o sistema de tempo real distribuído ARCOS - Architecture for Control and Supervision), uma plataforma baseada em componentes de tempo real para aplicações de controle e supervisão. O sistema de reconfiguração dinâmica proposto foi validado através de uma aplicação de controle veicular onde um controlador com defeito é substituído em tempo de execução por uma réplica. O impacto da reconfiguração, analisada a partir de métricas fim-a-fim de qualidade de controle, se mostrou adequado para a aplicação em questão.*

1. Introdução

A crescente disseminação e complexidade dos sistemas de tempo real demandam a utilização de tecnologias e padrões de projeto que minorem seu custo, tarefas de manutenção e tempo de desenvolvimento, sem negligenciar o atendimento correto de seus requisitos de confiabilidade. Aliado a esse fato, o emprego do *software* no controle de aplicações críticas (*eg*, controle de tráfego, sistemas de energia e telecomunicações) tem evidenciado a necessidade de mecanismos que aprimorem a qualidade de confiança no funcionamento (*dependability*) de tais sistemas, garantindo seu funcionamento correto, mesmo que de forma degradada, durante o maior período de tempo possível, sob pena de acarretar graves prejuízos financeiros ou humanos. A ocorrência de falhas, modificações nas condições do ambiente, obsolescência do software, redução substancial do desempenho dos processos ou da rede de comunicação entre equipamentos são alguns dos entraves que afetam as aplicações nesse cenário.

Uma forma de contornar a ocorrência de falhas consiste no uso de políticas de reconfiguração dinâmica (em tempo de execução), específicas às aplicações, que lhes permitam adaptação às novas condições do ambiente com a preservação das garantias temporais e funcionais previamente acordadas. A substituição eficiente de elementos faltosos é, portanto, essencial no contexto de sistemas de tempo-real. Além disso, políticas adequadas de reconfiguração possibilitam a concepção de procedimentos de manutenção corretiva e evolutiva que podem melhorar consideravelmente a disponibilidade do sistema. Nesse sentido, o uso da tecnologia de componentes de software tem se mostrado promissora.

Alguns exemplos notórios de tecnologia de componentes incluem Microsoft COM+, .NET [IIOP.Net 2004], *Sun Enterprise Java Beans* [Sun MicroSystems 2006] e OMG CORBA *Component Model* [Object Management Group 2006]. Como mencionado, as funcionalidades providas por essa tecnologia facilitam a construção de aplicações com requisitos de reconfiguração dinâmica, o que favoreceu também o surgimento de diversos *frameworks* com esse intuito [Batista, Joolia and Coulson 2005, Chan e Wu 2002, Hillman e Warren 2004, Rasche e Polze 2005, Schneider, Picioroagă, e Brinkschulte 2004]. Apesar de tal diversidade, poucos destes *frameworks* de desenvolvimento visam atender às aplicações de controle e supervisão em ambientes industriais de tempo real. Aplicações de controle e supervisão caracterizam-se, principalmente, pela existência de um ciclo de controle, no qual os dados obtidos através do sensoriamento da planta são enviados a um controlador. O controlador, por sua vez, efetua cálculos baseados nos valores medidos e o valor desejado (*setpoint*) das variáveis controladas e determina a ação a ser executada na planta, por meio dos atuadores disponíveis. Tal ciclo de controle gera uma restrição temporal adicional que deve ser satisfeita pelo mecanismo de reconfiguração dinâmica. Outra seja, nesse cenário, é importante que a reconfiguração aconteça de forma a minimizar a interferências danosas ao funcionamento do sistema controlado.

Este trabalho descreve a concepção e implementação de um serviço de reconfiguração dinâmica no contexto de sistemas de tempo real distribuídos. Na abordagem adotada, tanto os elementos constituintes do serviço quanto os da aplicação são implementados como componentes, beneficiando-se de suas características de reusabilidade e encapsulamento inerentes, favorecendo dessa forma a adaptação da aplicação às novas necessidades do ambiente de execução, bem como dos componentes do serviço de reconfiguração. Além disso, a adoção do CIAO (*Component-Integrated ACE ORB*) [Wang 2004], uma adaptação do CORBA *Component Mode*l (CCM) visando atendimento às aplicações distribuídas de tempo real, torna possível ao desenvolvedor da aplicação aproveitar os mecanismos existentes na plataforma utilizada para imprimir maior grau de previsibilidade ao sistema. O serviço apresentado provê primitivas e mecanismos que permitem a implementação de políticas específicas de tolerância a falhas que proporcionam a melhoria da disponibilidade das aplicações. A fim de verificar a adequação do modelo proposto, um protótipo do serviço de reconfiguração foi implementado e avaliado através da análise de métricas de desempenho relativas a uma aplicação de controle veicular, onde um controlador faltoso é substituído por uma réplica usando o mecanismo de reconfiguração dinâmica. Os resultados experimentais obtidos ressaltam a adequação do serviço de reconfiguração à aplicação utilizada.

O serviço de reconfiguração proposto foi concebido para ser integrado ao *framework* ARCOS (ARquitetura de COntrole e Supervisão) [Andrade e Macêdo 2005, Andrade et al 2006, Andrade e Macedo 2007, Macêdo et al 2004] cujo objetivo é prover uma plataforma reutilizável e extensível para sistemas de supervisão e controle, abordando

aspectos como interoperabilidade e mecanismos para garantias temporais e cuja plataforma também utiliza o CIAO.

O restante desse artigo está estruturado da seguinte maneira. A Seção 2 fornece uma visão geral do serviço de reconfiguração de componentes. A Seção 3 apresenta uma aplicação-exemplo e aspectos de sua implementação no serviço de reconfiguração proposto. A Seção 4 descreve a avaliação experimental da aplicação-exemplo através de alguns cenários de reconfiguração. Os principais trabalhos relacionados ao tema deste artigo são discutidos na Seção 5 e, por fim, a Seção 6 conclui o artigo apresentando considerações finais e perspectivas de trabalhos futuros.

2. Serviço de reconfiguração dinâmica de componentes

O serviço de reconfiguração dinâmica, proposto neste trabalho, objetiva prover mecanismos que permitam modificar a configuração do sistema em tempo de execução, preservando a consistência. Nas seções seguintes, apresentaremos o modelo do sistema e uma visão geral sobre a arquitetura do serviço de reconfiguração proposto.

2.1 Modelo do sistema

O sistema é formado por componentes conectados através de portas. Através das portas, o componente especifica quais os serviços que provê aos demais, bem como suas dependências em relação aos outros componentes. A interação dos componentes é feita através da conexão de portas de provisão, denominadas facetas, com portas de recepção de serviços, denominadas receptáculos. Para uso dos serviços providos por outro componente, é preciso haver uma conexão entre o receptáculo do componente e a faceta do componente provedor. Cada tipo de componente é gerenciado por uma implementação da interface *Home*, que provê operações para criação e localização de componentes. Assume-se que as falhas dos componentes são do tipo *fail-stop*.

2.2 Arquitetura do serviço de reconfiguração

A arquitetura proposta do serviço de reconfiguração é dividida em três elementos: *Reconfiguration Manager* (RM), *Consistency Manager* (CM), e *Deployment Manager* (DM). O *Reconfiguration Manager* é o principal responsável e coordenador do procedimento de reconfiguração. Para efetuar uma política de reconfiguração, este utiliza o *Consistency Manager* e o *Deployment Manager*. O primeiro tem por objetivo preservar a consistência global do estado do sistema, em especial, conduzindo os componentes a um estado consistente através do (des)bloqueio controlado de suas conexões e de sua (re)inicialização. O *Deployment Manager*, por sua vez, é utilizado para criar e destruir componentes ou as conexões entre componentes. A Figura 1 ilustra uma inter-relação simplificada entre estes elementos, descritos nas seções seguintes.

2.2.1 Reconfiguration Manager (RM)

O RM é o componente responsável por gerenciar a reconfiguração do sistema. Através deste, é possível criar e remover componentes e conexões, além de transferir estado entre componentes. As operações de reconfiguração possíveis através do serviço de reconfiguração são: criação e remoção de componentes, transferência de estado entre eles e (re)conexão de suas portas. É possível submeter cada operação de reconfiguração separadamente ou através de um *script* de reconfiguração com operações no formato XML.

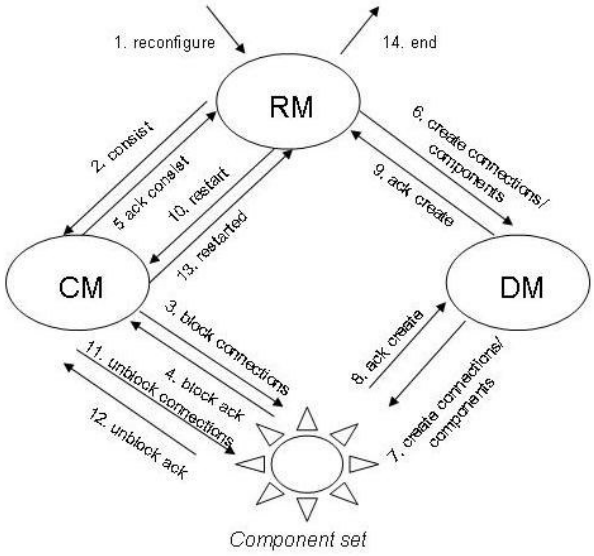


Figura 1. Visão geral do serviço de reconfiguração

Conforme ilustrado na Figura 1, a reconfiguração inicia através da ativação do método *reconfigure*, que recebe como parâmetro um conjunto de operações de reconfiguração. Caso a aplicação tenha definido uma política de consistência através de um CM, este é utilizado para conduzir o sistema a um estado adequado para a reconfiguração (Seção 2.2.2). Por exemplo, no cenário explorado na seção 3 (controle de velocidade veicular), considera-se que o estado de consistência foi alcançado através da abordagem sugerida em [Wermelinger 1999]. Essa abordagem bloqueia somente as conexões necessárias à reconfiguração, o que favorece a continuidade de outros serviços providos pelo componente, aumentando a disponibilidade da aplicação. Em seguida, o RM se conecta ao DM para acionar as mudanças estruturais da aplicação, descritas na Seção 2.2.3.

O RM permite a criação de políticas de reconfiguração através da definição de eventos e ações correspondentes de reconfiguração, que são acionadas quando da ocorrência de eventos. Para facilitar a escrita de políticas de reconfiguração nesse ambiente, foi concebida uma linguagem específica que permite descrever cenários de reconfiguração de forma mais simples e clara do que em XML. A linguagem é baseada no paradigma de comandos guardados, no qual a ocorrência de eventos ou avaliação verdadeira de predicados dispara a execução de blocos de comandos. Um exemplo de cenário de reconfiguração é descrito na Seção 3.1.

2.2.2 Consistency Manager (CM)

O CM é o componente responsável por conduzir o sistema a um estado apropriado para a realização da reconfiguração. Fundamentalmente, sua atuação concerne o envio de ordens (implementadas através de eventos) aos componentes da aplicação participantes do procedimento de reconfiguração. Para isso, os componentes envolvidos na ação de reconfiguração devem utilizar portas receptoras de eventos específicos para este fim. Estas ordens envolvem o bloqueio e desbloqueio das conexões destes componentes.

Para alcançar um estado consistente de reconfiguração, o CM mantém o registro das conexões da aplicação e se comunica com os componentes de forma assíncrona, solicitando o bloqueio de conexões, quando devido. Como ilustra a Figura 1, após ser

ativado pelo RM (*2. consist*), o CM efetua o bloqueio das conexões envolvidas (*3. block connections*). Por sua vez, o componente informa ao CM quando a solicitação de bloqueio for atendida (*4. block ack*). Para evitar *deadlock*, as solicitações de bloqueios e desbloqueios consideram a dependência entre as conexões; ou seja, uma conexão é bloqueada somente quando as conexões que dela dependem já estiverem bloqueadas. Do mesmo modo, uma conexão é desbloqueada somente quando as conexões das quais depende já o tiverem sido.

Uma vez que todas as solicitações de bloqueio tenham sido atendidas, o CM informa a situação ao RM (*5. ack*), que prossegue com a reconfiguração interagindo com o DM (Seção 2.2.3). Finda esta interação, o RM solicita ao CM que desbloqueie as conexões (*10. restart*) que, por sua vez, notifica os componentes a esse respeito (*11. unblock connections*) de modo que o sistema possa retornar ao seu estado normal de funcionamento.

A depender da abordagem de consistência adotada, o CM pode apresentar comportamentos distintos. Por exemplo, utilizando-se a abordagem de preservação de consistência descrita por [Kramer and Magee 1990], todos os nós que participam direta ou indiretamente de conexões a serem removidas devem ser conduzidos ao estado passivo, no qual apenas atendem requisições de cuja execução dependa a completude de outras. Já na abordagem orientada a conexões de Wermelinger [Wermelinger 1999], o estado para reconfiguração é alcançado através do bloqueio das conexões que serão removidas, obedecendo-se uma ordem dada pela dependência entre elas. Observa-se, portanto, que a determinação dos elementos a terem seu comportamento alterado e a ordem em que isso é realizado diverge a depender da estratégia empregada. Através da arquitetura apresentada no presente trabalho, é possível implementar *Consistency Managers* com comportamentos distintos e conectá-los ao *Reconfiguration Manager*, *Deployment Manager* e componentes da aplicação. Obviamente, estes últimos devem estar preparados para interagir com o CM através da provisão de portas de recepção e de emissão de eventos adequados à estratégia adotada.

2.2.3 Deployment Manager (DM)

O *Deployment Manager* é responsável pela criação, remoção de conexões e transferência de estado entre componentes. Estas funcionalidades são disponibilizadas através de uma interface que provê, entre outras, as seguintes operações: *Create/Remove Component*, *Create/Remove Connection* e *Transfer state*. A operação *Create component* recebe como parâmetro um identificador único para o componente, o nó em que o componente deve ser instanciado e o tipo do componente. A remoção do componente é feita através do fornecimento do seu identificador. As conexões são criadas através do comando *Create connection*, para o qual deve ser informado: o identificador da conexão e os componentes com respectivas facetas e receptáculos a serem conectados. A remoção de uma conexão requer apenas o identificador da conexão. Por fim, a operação *Transfer state*: efetua a transferência de estado entre componentes de mesmo tipo. Para isso, devem ser informados os identificadores dos componentes fonte e destino.

3. Estudo de caso: Reconfiguração dinâmica de uma aplicação de controle de velocidade veicular para substituição de um controlador falho

Para validação da abordagem, utilizamos o serviço de reconfiguração para gerenciar a falha de um dos componentes de uma aplicação de controle veicular. Nesse cenário, a detecção da falha de um controlador dispara o procedimento de substituição desse componente por uma réplica equivalente. Este serviço foi implementado no CIAO [Wang

2004], mesmo *middleware* com enfoque em aplicações distribuídas de tempo-real utilizado para a implementação do ARCOS. A seguir detalhamos as características principais dessa aplicação.

Na aplicação de controle veicular, o motorista informa a velocidade em que deseja trafegar e o sistema controla a atuação necessária a ser efetuada no acelerador do veículo para atingir esse objetivo. O comportamento físico da simulação do veículo é caracterizado pelas equações [Pont 2001]:

$$Accel = (Throttle * ENGINE\_POWER - (FRIC * Old\_speed)) / MASS \quad (1)$$

$$Dist = Old\_speed + Accel * (1/SAMPLE\_RATE) \quad (2)$$

$$Speed = SQRT ((Old\_speed)^2 + 2 * Accel * Dist) \quad (3)$$

O veículo apresenta propriedades de potência do motor, coeficiente de fricção e massa que são representadas pelas constantes ENGINE_POWER, FRIC e MASS. Informados a velocidade corrente (*Old_speed*) e o valor da atuação no acelerador (*Throttle*), se obtém a velocidade do veículo (*Speed*). A taxa de amostragem utilizada pelo controlador é representada pela variável SAMPLE_RATE.

Esse modelo serviu de base para implementação de um controlador PID (Proporcional-Integrativo-Derivativo) [Ogata 2001] que atua diretamente no acelerador com base no erro mensurado entre a velocidade atual e a velocidade desejada (*setpoint*). A calibração do valor calculado na saída do controlador depende da ponderação de três parâmetros essenciais: ganho proporcional (*kp*), ganho integral (*ki*) e ganho derivativo (*kd*). O termo proporcional enfatiza a reação do controlador aos erros momentâneos. O termo integral considera a reação segundo o somatório dos erros (histórico). Por fim, o termo derivativo determina a reação em função da taxa de modificação do erro mensurado (derivada). A sintonia desses parâmetros visa atender aos requisitos da aplicação tais como tempo de subida, tempo de estabilização e *overshoot* [Ogata 2001].

A implementação dos componentes dessa aplicação está apresentada na Figura 2. O *Gerador de Pulso* produz pulsos, como referências temporais, para o *Sensor* de velocidade na taxa de amostragem especificada pela aplicação de controle, cujo valor padrão é de 2 Hz. A cada pulso recebido, o *Sensor* obtém o valor da velocidade atual e a repassa para o *Controlador1*. Este, por sua vez, calcula o novo valor de atuação no acelerador de acordo com o erro entre o valor medido e o valor desejado para a velocidade.

A falha do *Controlador1* é tolerada através de uma variante do esquema de replicação semi-ativa, sendo que o estado da réplica (*Controlador2*) é atualizado a cada novo cálculo do valor de atuação. O *Controlador1* repassa os valores de atuação ao *Atuador*, que atualiza os parâmetros correspondentes no objeto controlado, representado pelo componente *Veículo*. A detecção de falhas é implementada através da conexão utilizada para transferência de estado existente entre o *Controlador2* e o *Controlador1*. Caso o intervalo entre duas operações de atualização de estado ultrapasse determinado valor especificado no *Controlador2*, este produz um evento de notificação de falha ao *Reconfiguration Manager*. Este último, por sua vez, aciona a reconfiguração do sistema, de acordo com a política especificada. Assume-se, que na ocorrência de falhas, o *Controlador1* possui um comportamento do tipo *fail-stop*.

As equações (1), (2) e (3) são utilizadas pelo *Sensor* para calcular a velocidade atualizada do *Veículo*. Desse modo, ao receber o pulso do *Gerador de Pulso*, o *Sensor* se comunica com o *Veículo* obtendo o valor atual da variável *throttle*. Este valor é substituído

na equação (1) para calcular a aceleração do veículo, que é dependente da atuação no acelerador e da velocidade anterior. Conhecendo-se a aceleração, a distância que o veículo percorreu durante o tempo transcorrido entre uma medição e outra é obtida através da equação (2). Finalmente, a equação (3) é empregada para o cálculo da velocidade atual do *Veículo*. Este valor é o que é enviado ao *Controlador1* para ser comparado com o *setpoint*, para correção da atuação, caso necessário.

A estratégia de preservação de consistência implementada foi a de bloqueio de conexões [Wermelinger 1999], mencionada na seção 2.2.2.

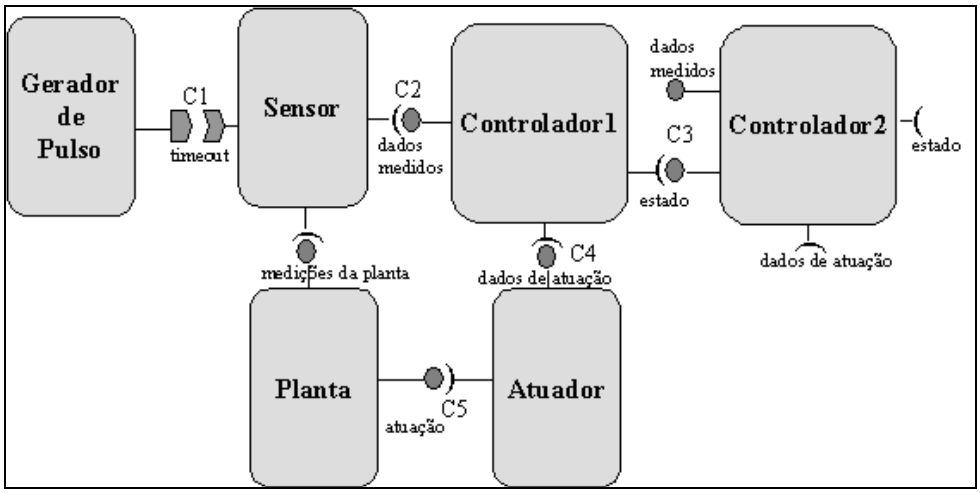


Figura 2. Componentes da aplicação de controle de velocidade veicular

3.1 Exemplo de cenário de reconfiguração

Esta seção apresenta um cenário de reconfiguração que efetua a substituição de um controlador falho através da reorganização das conexões das quais ele participa. A política de reconfiguração é escrita numa linguagem específica que fornece abstrações de alto nível para o desenvolvedor da política de reconfiguração e permite a geração automática de código para XML. A política de reconfiguração de substituição de um controlador falho é apresentada no trecho de código da Figura 3.

```
01 (Controlador1.failed) => {
02    destroy connection C2, C3, C4
03    create connection C6 : Sensor.r1 -> Controlador2.f1
04    create connection C7 : Controlador2.r1 ->  Atuador.f1  }
```

Figura 3. Política de reconfiguração de um controlador falho

A notificação da falha do componente *Controlador1* ao *Reconfiguration Manager*, expressa através do evento `Controlador1.failed`, aciona a remoção das conexões originais e efetua a conexão do novo controlador aos componentes existentes. Em primeiro lugar, o *script* remove as conexões originais (*C2, C3* e *C4*) através do comando `destroy` (linha 2), que atua sobre uma lista de conexões existentes. Nas linhas 03 e 04 são estabelecidas duas conexões (*C6* e *C7*) entre o novo controlador e os componentes pertinentes (*Sensor* e *Atuador*). Os descritores f1 e r1 representam as facetas e receptáculos dos componentes envolvidos no procedimento de conexão. A título de exemplo, o trecho de código da Figura 4 representa o código XML gerado para o comando de criação da conexão *C6* entre o componente *Controlador2* e o Sensor (linha 3) da Figura 3.

```
01 <command>
02   <createConnection>
03     <id>C6</id>
04     <type>SIMPLEXRECEPTACLE_FACET</type>
05       <initiator>
06         <portName>r1</portName>
07         <instance>Sensor</instance>
08       </initiator>
09       <receptor>
10         <portName>f1</portName>
11         <instance>Controlador2</instance>
12       </receptor>
13   </createConnection>
14 </command>
```

Figura 4. Código XML referente à criação do Controlador3 (linha 3 da Figura 3)

A utilização de uma linguagem específica facilita a verificação automática de propriedades importantes no contexto do domínio de reconfiguração. Por exemplo, no cenário de falha exposto, espera-se que o comando `destroy` atue somente nas conexões nas quais haja participação do componente falho (*C2*, *C3* e *C4*, no caso), pois não faz sentido, além de ser extremamente indesejável, afetar as conexões de outros componentes nesse contexto. Tal verificação é facilmente realizável através de análise simples dos comandos do *script* de reconfiguração.

4. Avaliação de desempenho

O experimento foi conduzido em um computador com processador Pentium IV 2.66GHZ, 512MB de memória RAM e sistema operacional Debian Linux kernel 2.6. Os valores das propriedades do veículo e de configuração do controlador foram extraídos de [Andrade 2006]: ENGINE_POWER=5000, FRIC=50, SAMPLE_RATE=2Hz, termo proporcional do controlador PID = 0.05, Termo derivativo do controlador PID = 0, termo integral do controlador PID = 0. Nos experimentos efetuados, o *script* de reconfiguração submetido ao *Reconfiguration Manager* continha os seguintes comandos: remover as conexões *C2*, *C3* e *C4*, remover o controlador original, criar conexão *C6* entre o novo controlador e o sensor e, por fim, criar conexão *C7* entre o atuador e o novo controlador. Os tempos obtidos são relativos ao início da reconfiguração pelo RM, após a falha ter sido notificada pelo *Controlador2*.

4.1 Métricas de desempenho

Para a aplicação veicular, o aspecto principal na avaliação do controlador consiste em verificar sua capacidade em manter a velocidade próxima ao valor desejado. Para este fim, três métricas de avaliação foram consideradas [Ogata 2001]: tempo de subida, tempo de estabilização e *overshooting*. O tempo de subida (*raise time*) é o tempo necessário para o valor atual (resposta) alcançar uma faixa percentual do valor desejado. Três faixas de valores são normalmente utilizadas: 10% a 90%, 5% a 95% e 0% a 100%. O tempo de estabilização (*settling time*) é o tempo necessário para a curva de resposta alcançar e permanecer dentro de um intervalo em torno do valor final. Este intervalo é especificado por uma porcentagem absoluta do valor final (geralmente entre 2% a 5%). A última métrica é o percentual máximo de *overshoot* que corresponde ao valor máximo obtido pela curva de resposta.

Para avaliar o impacto da reconfiguração no desempenho da aplicação durante a reconfiguração foram definidos três cenários: reconfiguração simples; reconfiguração com modificação do *setpoint* e reconfiguração com simulação de perturbação, tendo sido a

reconfiguração acionada após a estabilização da velocidade. Para cada cenário, foram obtidas as curvas de resposta da velocidade do veículo, com e sem o procedimento de reconfiguração, de modo a avaliar o impacto da reconfiguração no desempenho da aplicação de controle. Estes cenários de avaliação são descritos nas seções seguintes.

4.2 Cenário 1: Reconfiguração simples

Este cenário consiste na aplicação de reconfiguração (através da falha forçada do *Controlador1*) após a estabilização do valor da velocidade do veículo em 40 km/h. A Figura 6 descreve a curva de evolução da velocidade do veículo na execução de reconfiguração aos 59s; portanto, após atingida a estabilidade da velocidade. A proximidade das curvas (a diferença é praticamente imperceptível no gráfico) de evolução da velocidade nas situações com a reconfiguração e sem a reconfiguração atesta a ausência de impacto significativo na aplicação.

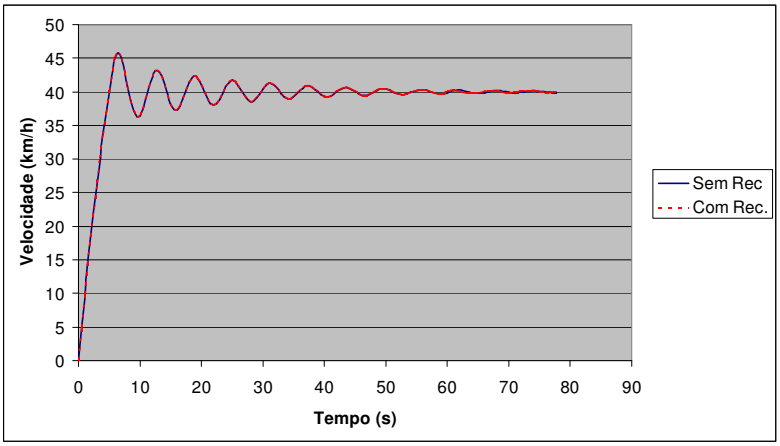


Figura 6. Evolução do Controlador PID de velocidade com reconfiguração aos 59s

Observaram-se os mesmos valores para as propriedades do sistema controlado com e sem a reconfiguração: tempo de subida ≈ 5 s, tempo de estabilização ≈ 22 s (considerando uma variação em torno de 5% do valor final) e *overshoot* ≈ 14 %. Os custos referentes às atividades de reconfiguração são: intervalo entre a solicitação de bloqueio da conexão e seu bloqueio efetivo 460 ms, o intervalo entre a solicitação de desbloqueio da conexão e seu desbloqueio efetivo: ≈ 500 ms.

4.3 Cenário 2: Reconfiguração com modificação de *setpoint*

Este cenário consiste na aplicação de reconfiguração após a estabilização do valor da velocidade do veículo em 40 km/h, e posterior mudança de *setpoint* para 70 km/h, seguida de reconfiguração aos 100 s. Assim como no cenário anterior, a Figura 7 ilustra diferença desprezível entre os tempos na ausência e presença da reconfiguração.

A Tabela 1 apresenta os valores do tempo de subida, tempo de estabilização e *overshoot* quando da modificação do *setpoint*. Vale ressaltar que os valores para o *setpoint* em 40 km/h são os mesmos do ambiente de testes de reconfiguração simples (cenário 1).

Setpoint	Tempo de subida	Tempo de estabilização	*Overshoot*
40 km/h	5s	22s	14%
70 km/h	10s	9s	4%

Tabela 1. Métricas de desempenho do controlador PID com modificação de *setpoint*

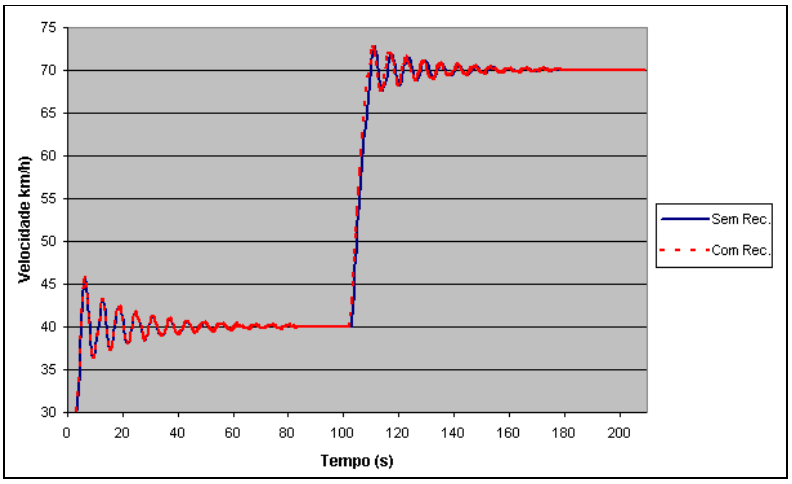


Figura 7. Evolução do Controlador PID com alteração da velocidade de 40km/h para 70km/h aos 100 segundos

4.4 Cenário 3: Reconfiguração perturbação simulada

Este cenário consiste na aplicação de reconfiguração após a estabilização do valor da velocidade do veículo em 40 km/h e posterior perturbação ao veículo através da simulação de rajada de vento em duas situações distintas. Na primeira situação, o valor da variável *throttle* foi alterado para 1.5 no instante 100s, simulando a ocorrência de vento incidente na traseira do veículo, ilustrado pela Figura 9. Logo após a perturbação, foi submetido o *script* de reconfiguração. Nesse caso, o sistema leva aproximadamente 2 s para alcançar o *setpoint*, ou seja, tanto com e sem reconfiguração, o tempo de subida é de aproximadamente 2 s. Em ambos os casos, o tempo de estabilização após a perturbação é de por volta de 4s e o *overshoot* é de aproximadamente 14%.

Na segunda situação, no instante 30 s, simulou-se a incidência de vento na dianteira do veículo modificando bruscamente o valor da variável *throttle* para 0,1. Após a perturbação (seguida imediatamente pela reconfiguração), o tempo de subida teve o valor de aproximadamente de 0,4 s. Observou-se que, após a perturbação, em momento algum a variação da velocidade alcançou valor inferior ou superior a 5% do *setpoint*. O *overshoot* ficou em torno de 4 %.

4.5 Discussão

Com base nos experimentos apresentados, observa-se que as ações de reconfiguração não tiveram impacto significativo no desempenho do controle do veículo. No entanto, a reconfiguração influi no comportamento do sistema, uma vez que tem como conseqüência o bloqueio das conexões entre o *Sensor* e o *Controlador* e entre o *Controlador* e *Atuador*. Por esse motivo, durante o tempo de reconfiguração, o *Sensor* não atualiza o *Controlador* com a velocidade do veículo, fazendo com que o valor da atuação permaneça o mesmo nesse período.

No experimento 1, uma vez que a reconfiguração se dá em um momento de estabilidade do sistema, sem influência de eventos externos, era esperado que ela tivesse pouco impacto no comportamento do mesmo.

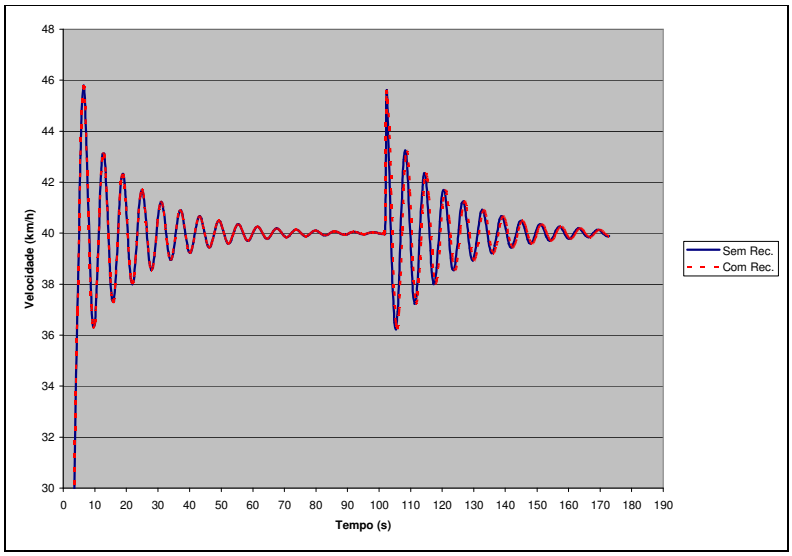


Figura 9. Evolução do Controlador PID com simulação de rajada de vento na traseira do veículo no instante 100 s

Nos cenários 2 e 3, a reconfiguração foi acionada imediatamente após a alteração do *setpoint* e da ocorrência de perturbação, o que poderia fazer com que a resposta do sistema a esses eventos fosse prejudicada pelo atraso causado pela reconfiguração. Entretanto, conforme observado nos gráficos e nas métricas do sistema, não houve influência significativa do tempo de reconfiguração no desempenho do mesmo. Tal comportamento pode ser atribuído ao fato de o intervalo de tempo em que as conexões do sistema permanecem bloqueadas ser em torno de 100 ms. Além disso, a taxa de amostragem variou entre 100 e 500 ms, o que possibilita ao *Controlador* corrigir a velocidade do *Veículo*, no intervalo de tempo relativamente curto. Espera-se que em aplicações em que seja necessário efetuar reconfigurações mais demoradas (por exemplo, com mais operações de re-conexão), a resposta do sistema a uma alteração de *setpoint* ou distúrbio externo seja mais prejudicada do que o observado nos experimentos apresentados.

5. Trabalhos Relacionados

Diversos trabalhos anteriores versaram sobre a reconfiguração dinâmica de componentes. Relacionamos, a seguir, alguns dos trabalhos relevantes na área.

Bidan et al (1998) propõem um serviço de reconfiguração dinâmica baseado em CORBA que permite criar e remover objetos e conexões, desde que inexistam chamadas de RPC pendentes entre os objetos envolvidos nas conexões que serão excluídas. Em [Chan e Wu 2002] é definido um modelo próprio de componentes denominado ComponentGOP, implementado sobre CORBA, cuja arquitetura é representada por grafos lógicos. A reconfiguração é comandada através da utilização de primitivas para atualização de grafos. OpenRec [Hillman e Warren 2004] é um *framework* para reconfiguração dinâmica que funciona sobre o modelo de componentes desenvolvido pelos autores. É definido um algoritmo básico de reconfiguração que deve ser especializado para atender aos requisitos de consistência da aplicação. Em [Rasche and Polze 2005] é apresentada uma infra-estrutura de reconfiguração em plataformas .NET que permite a criação, remoção e migração de componentes e utiliza a abordagem de Wermelinger [Wermelinger 1999].

LuaSpace é um ambiente para reconfiguração dinâmica de aplicações baseadas em componentes CORBA [Batista, Cerqueira e Rodriguez 2003]. Nessa abordagem, os objetos são acessados através de *proxies* que intermedeiam sua comunicação com o cliente. A depender da configuração da aplicação, o *proxy* encaminha a requisição ao objeto adequado através da utilização dos mecanismos de Repositório de Interfaces e Interface de Invocação Dinâmica do CORBA.

O SwapCIAO [Balasubramanian et al 2005] é uma extensão do middleware CIAO (*Component Integrated ACE ORB*) [Wang 2004] que permite atualizar a implementação de um componente em tempo de execução. Para tanto, dispõe de mecanismos para redirecionar os clientes de um componente existente para uma nova e atualizada implementação do mesmo. Plastik [Batista, Joolia and Coulson 2005] é um *meta-framework* que provê reconfiguração dinâmica através da combinação de uma linguagem de descrição arquitetural (ADL) com um modelo de componentes reflexivo chamado OpenCOM. Plastik fornece dois tipos de reconfiguração: programada e *ad-hoc*. A reconfiguração programada é feita através da especificação de comandos do tipo "predicado-ação", de modo similar ao nosso trabalho, ao passo que a reconfiguração *ad hoc* pode ser realizada através de *scripts* em ADL ou de comandos do OpenCOM.

Em [Hofmeister and Purtilo 1993] a aplicação é estruturada na forma de módulos nos quais podem ser definidos pontos de reconfiguração. Tais pontos indicam quando a reconfiguração é segura e qual é o estado da aplicação. O código fonte do programa é transformado através da adição dos comandos necessários para: (i) postergar a reconfiguração até o momento apropriado, (ii) organizar e enviar o estado, (iii) instalar o estado, restaurar a pilha de registro de ativação quando necessário, (iv) reiniciar a execução no lugar apropriado.

Em [Schneider, Picioroagă, and Brinkschulte 2004] é definido um serviço de reconfiguração intergradoao *middleware* OSA+ (*Open System Architecture*). Os serviços se comunicam através de tarefas (*jobs*) , onde uma tarefa é um par formado por uma ordem e um resultado e a reconfiguração é efetuada através da substituição ou movimentação de serviços, com e sem transferência de estado.

Apesar de disporem de ações de reconfiguração similares às apresentas neste trabalho, em [Bidan et al 1998] o modelo de sistema empregado é o de objetos definido pelo CORBA. No ComponentGOP [Chan e Wu 2002] e no OpenRec [Hilman e Warren 2004] utiliza-se um modelo de componentes próprio, sendo que o do primeiro é baseado em CORBA.

Embora o Plastik [Batista, Joolia and Coulson 2005] se utilize de um modelo de componentes, seu foco também não se dá em aplicações com necessidade de garantias temporais. O trabalho apresentado em [Schneider, Picioroagă, and Brinkschulte 2004] é desenvolvido com base em um middleware com suporte a tempo-real, entretanto, o sistema é composto por serviços e jobs encaminhados a eles, não fazendo uso do modelo de componentes e, portanto, se restringindo à substituição de serviços.

O modelo de componentes e middleware utilizados no presente trabalho são os mesmos empregados no SwapCIAO [Balasubramanian et al 2005]. Entretanto, as funcionalidades previstas no SwapCIAO ainda não estão integradas ao CIAO, e não há menção de outras formas de reconfiguração além da substituição de implementações de componentes previamente existentes, o que possibilita uma reconfiguração limitada.

No trabalho descrito em [Hofmeister and Purtilo 1993] os pontos de reconfiguração são previamente especificados e codificados no programa, o que prejudica

a flexibilidade da reconfiguração. Além disso, não é demostrada aplicabilidade em sistemas de tempo-real. Em [Rasche e Polze 2005] os autores utilizam a .NET como plataforma para reconfiguração e avaliam o atendimento aos requisitos temporais de uma aplicação através do cálculo do seu tempo máximo de bloqueio.

Além dos diferenciais apresentados, nossa abordagem separa os mecanismos de reconfiguração e consistência, o que possibilita a utilização de diferentes abordagens para esse fim. Outro aspecto importante é que a comunicação com o ambiente de execução é mediada por um componente genérico, o que facilita sua integração em plataformas de componente diversas. Além disso, o fato da estrutura do serviço de reconfiguração ser baseada em componentes favorece sua integração com outros serviços e, até mesmo, sua própria reconfiguração.

6. Considerações finais e perspectivas

Lidar adequadamente com as falhas de componentes de software em tempo de execução é de grande valia para sistemas com exigentes requisitos de disponibilidade, a exemplo de aplicações de controle em tempo real. Nesse sentido, serviços de reconfiguração dinâmica são ferramentas úteis na manutenção do desempenho da aplicação em níveis adequados ou, caso necessário, na sua adaptação de acordo com as novas condições do ambiente.

Este trabalho descreveu o projeto e a implementação de um serviço de reconfiguração no contexto de sistemas de tempo real distribuído. Os mecanismos implementados foram avaliados no contexto de uma aplicação representativa do domínio: controle veicular. A análise dos resultados experimentais, através de métricas de qualidade de controle, revelou o atendimento adequado aos requisitos de desempenho da aplicação e o baixo impacto da implementação do serviço de reconfiguração.

Como trabalho futuro, avaliaremos o impacto dos mecanismos de reconfiguração dinâmica em plantas reais, não simuladas.

Referências

Andrade, S. Macêdo, R. (2005) "A Component-Based Real-Time Architecture for Distributed Supervision and Control Applications". In Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA2005) p. 15-22.

Andrade, S. S., Macêdo, R. J., Sa, A. S. and Santos, N. P. (2006) "Using Real-time Components to Construct Supervision and Control Applications", In: Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS 2006) Work in Progress Session.

Andrade, S., Macêdo R. (2007) "Engineering Components for Flexible and Interoperable Real-Time Distributed Supervision and Control Systems". In 12th IEEE Conference on Emerging Technologies and Factory Automation.

Andrade, S. S. (2006). "Sistemas Distribuídos de Supervisão e Controle Baseados em Componentes de Tempo Real". Dissertação de Mestrado. Universidade Federal da Bahia.

Balasubramanian, J., Natarajan B., Parsons, J., Schmidt, D. C. and Gokhale A. (2005) "Middleware Support for Dynamic Component Updating", In: Proceedings of the International Symposium on Distributed Objects and Applications (DOA).

Batista, T. V., Joolia A. and Coulson G. (2005) "Managing Dynamic Reconfiguration in Component-Based Systems", In: Proceedings of the 2nd European Workshop on Software Architecture.

Batista, T., Cerqueira, R. and Rodriguez N. (2003) "Enabling Reflection and Reconfiguration in CORBA", In: 2nd Workshop on Reflective and Adaptative

Middleware - ACM/IFIP/USENIX International Middleware Conference. pp 125 – 129.

Bidan, C., Issarny, V., Saridakis, T., Zarras, A. (1998) “A dynamic reconfiguration service for CORBA”, In: Proceedings of the IEEE International Conference on Configurable Distributed Systems.

Chan, A. S and Wu, G. 2002. “Architectural Level support for Dynamic Reconfiguration and Fault Tolerance in Component-Based Distributed Software”. In Proceedings of the 9th International Conference on Parallel and Distributed Systems. IEEE Computer Society, Washington, DC, 251.

Coulson, G., Blair, G., Clarke, M. and Parlavantzas, N. (2002) "The design of a configurable and reconfigurable middleware platform". Distributed Computing Journal. Volume 15, Number 2, p. 109-126.

Hofmeister, C. and Purtilo, J. (1993) “Dynamic reconfiguration in distributed systems: Adapting software modules for replacement”, In: Proceedings of the 13th International Conference on Distributed Computing Systems, p. 101--110.

Hillman J. and Warren I. (2004) “An Open Framework for Dynamic Reconfiguration”, In: 26th International Conference on Software Engineering (ICSE), pp. 594-603.

IIOP.NET Homepage. (2004) “.NET, CORBA and J2EE Interoperation”. http://iiop-net.sourceforge.net.

Kramer, J. and Magee, J. (1990) “The Evolving Philosophers Problem: Dynamic Change Management”, In: IEEE Transactions on Software Engineering. Volume 16, Issue 11.

Macêdo, R. J. A et al. (2004). "Tratando a Previsibilidade em Sistemas de Tempo Real Distribuídos: Especificação, Linguagens, Middleware e Mecanismos Básicos", In: 22º Simpósio Brasileiro de Redes de Computadores". Livro texto do minicurso, p. 105-163.

Object Management Group. (2004) “CORBA Specification”, http://www.omg.org/cgi-bin/doc?formal/04-03-01.

Object Management Group. (2006) “CORBA Component Model Specification”, http://www.omg.org/cgi-bin/doc?formal/06-04-01.

Ogata, K. (2001). “Modern Control Engineering”. 4th edition, Prentice-Hall, New Jersey.

Pont, M. G. (2001) “Patterns for time-triggered embedded systems”, Addison-Wesley Professional, 1st edition.

Rasche, A. and Polze, A. (2005) “Dynamic Reconfiguration of Component-based Real-time Software”, In: Proceedings of the 10th IEEE international Workshop on Object-Oriented Real-Time Dependable Systems.

Sans, Ricardo, Segarra, Miguel, Losert, Thomas, Bermejo, Julita, Arzén and Karl-Erik. (2003) “Engineering Handbook for CORBA-based Control Systems.” Madrid:Universidad Politécnica de Madrid.

Schneider, E., Picioroaga, F., and Brinkschulte, U. (2004) “Dynamic reconfiguration through OSA+, a real-time middleware”. In: Proceedings of the 1st Int. Doctoral Symp. on Middleware.

Sun MicroSystems (2006). Enterprise Java Beans 3.0. http://java.sun.com/products/ejb/docs.html.

Wang, N. (2004) “Composing Systemic Aspects Into Component-Oriented DOC Middleware”, PhD thesis, St. Louis: Washington University.

Wang, N., Gill, C. (2004) “Improving Real-Time System Configuration via a QoS-aware CORBA Component Model”, In: Proceedings of the 37th Hawaii International Conference on System Sciences.

Wermelinger, M. A. (1999) “Specification of software architecture reconfiguration”, Ph.D. thesis, Universidade Nova de Lisboa.

Uma Proposta para Reconfiguração Consistente no Nível Arquitetural

Jonivan Lisbôa[1], Orlando Loques[1]

[1]Instituto de Computação – Universidade Federal Fluminense (UFF)
Rua Passo da Pátria 153, Bloco E – 24210-240 – Niterói – RJ – Brasil
{jlisboa,loques}@ic.uff.br

Resumo. *Em aplicações adaptativas, a manutenção da consistência após uma adaptação está associada à manutenção de seus requisitos funcionais e não-funcionais, respeitando-se as dependências entre seus componentes. A consistência possui três aspectos: estado dos componentes, comunicação e configuração da aplicação. Este trabalho explora o último deles, propondo um modelo para manutenção da consistência da configuração no nível arquitetural, através do tratamento de falhas nas atividades básicas de configuração de componentes. O objetivo principal é evitar incoerências entre o modelo de alto nível utilizado e a configuração real, e assim não comprometer o uso de arquiteturas de software no projeto de aplicações.*

Abstract. *In adaptive applications, the maintenance of consistency after an adaptation is related to maintenance of both functional and non-functional requirements, respecting dependencies among their components. Consistency has three aspects: component state, communication, and application configuration. This work explores the last one, proposing a model for configuration's consistency maintenance at architectural level, by treating failures in basic component configuration activities. The main goal is to avoid incoherences between high level model and actual configuration, and so do not compromise the utilization of software architectures in application design.*

1. Introdução

Diversas aplicações computacionais modernas são projetadas para adaptar automaticamente sua configuração em tempo de execução, visando adquirir características como alta disponibilidade e tolerância a falhas. O desenvolvimento de aplicações adaptativas cresceu com o lançamento da idéia de Computação Autônoma (*Autonomic Computing*) [Kephart e Chess 2003], uma iniciativa que visa criar sistemas computacionais com a capacidade de auto-gerenciamento, sem a intervenção direta do ser humano. Este, como projetista, passa a ter o papel de definir políticas e regras gerais que servem como entrada para o processo de auto-gerenciamento, que envolve: auto-configuração (configuração automática de componentes), auto-correção (detecção automática de falhas), auto-otimização (controle de alocação de recursos para funcionamento ótimo) e auto-proteção (proteção contra ataques).

Adaptações na configuração ocorrem através da execução seqüencial de operações sobre o ciclo de vida de componentes – ativação, desativação e troca de ligações. Desde trabalhos primordiais sobre reconfiguração dinâmica [Kramer e Magee

1985], a manutenção da consistência da aplicação após uma adaptação é um problema relevante. A consistência está relacionada ao atendimento de todos os requisitos funcionais e não-funcionais definidos para a aplicação, respeitando-se as dependências existentes entre os componentes envolvidos na adaptação [Kon e Campbell 2000]. Em linhas gerais, o problema pode ser analisado sob três aspectos:

- A consistência de estado relaciona-se à manutenção do estado de computação apresentado pela aplicação no momento da adaptação;
- A consistência de comunicação relaciona-se com o tratamento de possíveis mensagens em trânsito entre componentes envolvidos na adaptação, sem prejuízo para a funcionalidade da aplicação;
- A consistência de configuração está ligada à manutenção de propriedades invariantes da aplicação como um todo. Após uma adaptação, o conjunto de componentes configurados deve continuar a atender adequadamente os requisitos definidos para a aplicação. Deve-se também manter a coerência entre o modelo de alto nível e a configuração real de componentes.

Em diversas abordagens citadas mais adiante, a consistência da configuração é um aspecto pouco explorado, por não existir na maioria delas a preocupação com a visão arquitetural da aplicação. Julgamos que isso é uma ferramenta importante para a obtenção de benefícios como análise comportamental em alto nível e separação entre interesses funcionais e não-funcionais. Esse fato serve como motivação para a proposta deste trabalho: apresentar um modelo para implementação de reconfiguração consistente no nível arquitetural. O modelo proposto apóia-se na detecção e tratamento de falhas nas atividades relacionadas à configuração de componentes, que possuem potencial para gerar inconsistências no modelo de alto nível. Espera-se assim manter a equivalência entre o modelo arquitetural utilizado e a configuração real, não se comprometendo o uso de arquiteturas de software e seus benefícios na fase de projeto da aplicação.

A seqüência deste artigo contém: a Seção 2, que apresenta paradigmas considerados para a elaboração da proposta; a Seção 3, que apresenta trabalhos correlatos sobre adaptação dinâmica; a Seção 4, que apresenta o modelo proposto; a Seção 5, que apresenta um exemplo para validação; e a Seção 6, que apresenta uma análise da proposta, conclusões e direções futuras.

2. Paradigmas considerados

O desenvolvimento de aplicações adaptativas vem convergindo para a utilização de determinadas técnicas e paradigmas, que foram considerados para o modelo proposto neste trabalho:

- Modelos de componentes que servem como uma abstração para descrever recursos de hardware e software e permitem o desenvolvimento de aplicações através da composição de blocos independentes. O desenvolvimento baseado em componentes é derivado da pesquisa e utilização na indústria da tecnologia de orientação a objetos, e ganhou força com a disseminação de plataformas como CORBA Component Model [OMG 2008], Enterprise Java Beans [Sun 2008], COM+/.Net [Microsoft 2008], dentre outras. Tais plataformas são responsáveis

por fornecer serviços básicos de gerenciamento de componentes, como controle de ciclo de vida, persistência, nomes e diretório, transações, introspecção, adaptação, e outros;

- Conceitos de Arquiteturas de Software [Shaw e Garlan 1996] para modelar em alto nível o comportamento de uma aplicação, em função de seus componentes e as interações existentes entre eles. Isso permite análises comportamentais na fase de projeto e separação clara entre interesses funcionais e não-funcionais, com a utilização de contratos entre componentes [Meyer 1992, Beugnard et al. 1999] para descrever configurações possíveis para a aplicação levando-se em conta requisitos de qualidade de serviço, e contratos de adaptação para definir regras de transição entre configurações;
- Configuração de recursos e adaptação da configuração em tempo de execução, através de *frameworks* programáveis que reúnem modelos arquiteturais e mecanismos de gerenciamento de componentes. A adaptação dinâmica apóia-se na capacidade dos *frameworks* de avaliar as condições do ambiente de execução de uma aplicação e confrontá-las com os requisitos funcionais e, principalmente, não-funcionais definidos para ela. Os *frameworks* de adaptação atuam sobre a plataforma de gerenciamento de componentes, de modo a controlar de forma dinâmica o seu ciclo de vida (ativação, desativação) e as ligações entre eles, permitindo a conexão e desconexão em tempo de execução.

3. Trabalhos relacionados

Diversos trabalhos apresentam *frameworks* para suporte ao gerenciamento automático de configuração. Dentre eles, podemos destacar: QuO [Loyall et al. 1998], Rainbow [Garlan et al. 2004], Q-CAD [Capra et al. 2005], Plastik [Batista et al. 2005], CR-RIO [Cardoso et al. 2006]. O funcionamento dos *frameworks* baseia-se no esquema "sentir-planejar-agir" dos sistemas de controle para automatizar a adaptação em aplicações baseadas em componentes, e apresentam como elementos principais:

- Um modelo para descrição em alto nível de tipos de componentes, configurações possíveis para a aplicação, condições para ativação de cada configuração e regras de transição entre configurações;
- Um sistema de informação de contexto, com monitoração de propriedades do ambiente através de sensores ligados aos componentes, e armazenamento em bases de dados apropriadas;
- Um gerenciador de configuração que toma decisões de acordo com o contexto observado e as condições previstas nos contratos, fazendo a aplicação reagir a variações observadas no ambiente de execução, através da intervenção no ciclo de vida de componentes e nas ligações entre eles.

Inicialmente, visa-se aplicar o modelo proposto neste trabalho no *framework* CR-RIO, que foi desenvolvido e vem sendo continuamente aprimorado pelo grupo de pesquisa ao qual os autores pertencem. Como diferencial em relação aos outros, o CR-RIO possibilita a obtenção de uma visão arquitetural da aplicação, com a descrição de arquiteturas e contratos. Mediante uma linguagem declarativa (CBabel), são descritos

tipos de componentes e instâncias, ligações permitidas entre eles e possibilidades de configuração considerando-se requisitos de qualidade.

A Figura 1 apresenta o esquema do *framework* CR-RIO, e seus elementos: sensores de monitoração de propriedades (*Resource Agents*); sistema de informação de contexto; registro de componentes validados, que serve como base de dados para obtenção dinâmica de referências a componentes; configurador, que atua sobre o ciclo de vida dos componentes executando os comandos de configuração; e o próprio elemento gerenciador. Alguns detalhes sobre sintaxe e semântica dos comandos da linguagem podem ser encontrados no exemplo descrito na Seção 5.

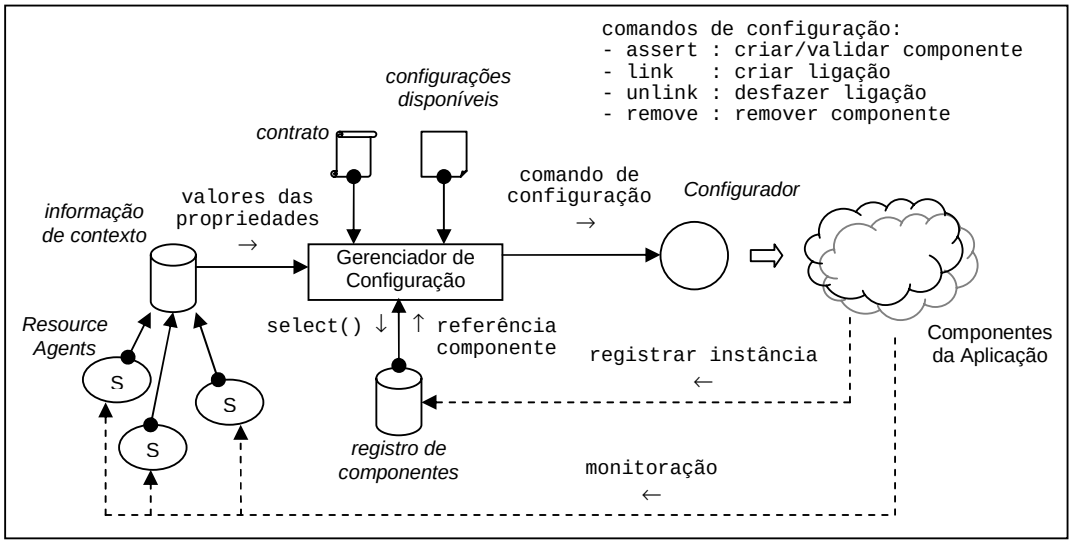


Figura 1. Esquema do *framework* CR-RIO

Um fato relevante observado nos trabalhos destacados é que as alterações na aplicação são impostas sem que seja verificada a ocorrência de falhas na configuração dos componentes. Isso possui potencial para causar inconsistências entre a representação de alto nível da aplicação (arquitetura) e a sua configuração real, pois falhas na ativação de um componente ou na substituição de alguma ligação farão com que a configuração real apresente um estado errôneo e não-sinalizado no nível mais alto. Para resolver esse problema, a idéia principal do modelo proposto é obter um retorno sobre o sucesso de cada atividade básica sobre o ciclo de vida dos componentes envolvidos em uma adaptação, para manter a coerência entre a representação de alto nível da aplicação e sua configuração real.

4. Proposta para Reconfiguração Consistente

O modelo proposto neste trabalho realiza o tratamento de possíveis fontes de inconsistência no processo de adaptação dinâmica, através de detecção de falhas ocorridas no nível mais baixo da execução da adaptação. As falhas detectadas são tratadas no nível arquitetural pelo mecanismo de gerenciamento de configuração, segundo algoritmo detalhado mais adiante. Para este trabalho, assume-se que existe um *framework* programável capaz de monitorar as condições do ambiente e tomar decisões

sobre adaptações dinâmicas na configuração da aplicação – no caso, o CR-RIO e um contrato para gerenciamento da configuração.

4.1. Idéia básica da proposta

Uma forma de se garantir uma reconfiguração consistente é fazer com que a atividade de adaptação possua um caráter transacional atômico: a seqüência de atividades básicas para se alterar a configuração da aplicação deve ser executada completamente para que a alteração seja bem sucedida. Segundo [Romanovsky 2001], o modelo tradicional de transações [Gray e Renter 1993, Lynch et al. 1993] possui algumas falhas que motivaram sua extensão, ou aprimoramento. Uma delas é a não existência de mecanismos aceitáveis de recuperação frente a situações contornáveis como, por exemplo, as exceções de software que podem acontecer na execução de uma adaptação. Uma forma de realizar tal recuperação é relaxar a propriedade de atomicidade de uma transação, de modo que seja possível tratar falhas que a invalidariam.

A proposta deste trabalho oferece um mecanismo para relaxamento da atomicidade na atividade de adaptação. As operações básicas de configuração de componentes são organizadas em um nível mais interno de execução, e são executadas de forma gradual. Caso não ocorra falha na execução de uma operação, ela é adicionada a um histórico de operações bem sucedidas. Uma falha na execução é sinalizada através do disparo de uma exceção gerada pelo configurador de componentes e capturada para tratamento pelo gerenciador de configuração.

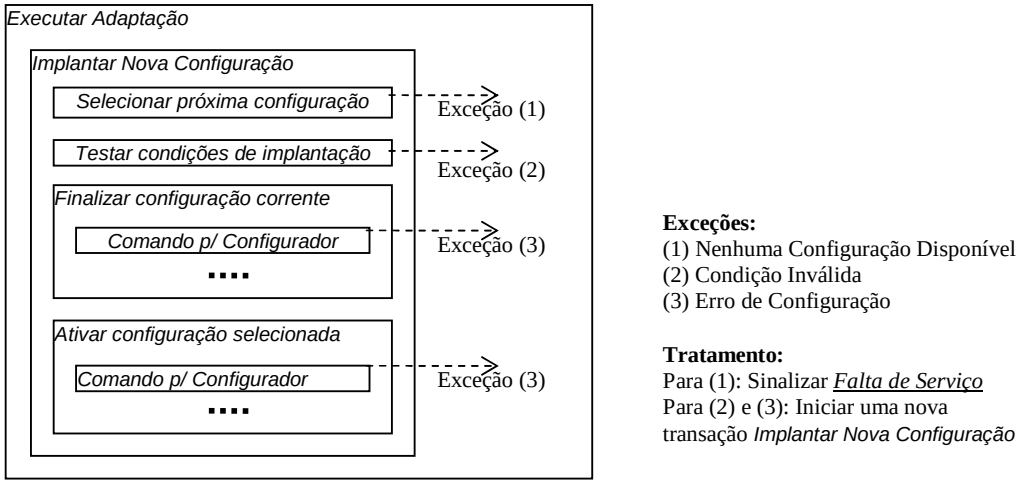


Figura 2. Organização das transações no modelo de adaptação proposto

A Figura 2 apresenta a organização em níveis das transações, representadas por retângulos, as possíveis exceções e o respectivo tratamento. A transação mais externa (*Executar Adaptação*) é iniciada nas situações em que uma adaptação é requerida: início da aplicação (primeira configuração); ocorrência de falha em algum componente da configuração corrente; ou, a configuração corrente deixa de atender aos requisitos de qualidade de serviço definidos em contrato, devido a variações observadas em propriedades monitoradas nos componentes configurados. A transação engloba uma outra (*Implantar Nova Configuração*), estruturada para executar uma seqüência de passos – também transações – para implantar uma configuração: (i) selecionar uma nova

configuração dentre as disponíveis; (ii) testar condições de implantação da nova configuração; (iii) finalizar a configuração corrente; (iv) ativar a nova configuração.

4.2. Execução da adaptação

A Figura 3 mostra um pseudocódigo para a transação do 2º. Nível (*Implantar Nova Configuração*), que contém a sequência de execução das transações mais internas (3º. Nível) que a compõem. A transação de 1º. Nível (*Executar Adaptação*) será ativada pela *thread* principal de execução do gerenciador de configuração, que monitora periodicamente a validade do contrato de adaptação. O pseudocódigo será útil para o entendimento dos exemplos apresentados na Seção 5. No código, **C** representa a configuração corrente, e **C´** representa uma nova configuração.

```
Transação do 2º. Nível
  ImplantarNovaConfiguração()
  Início
    C´ = SelecionarConfiguração()
    TestarPerfil(C´)
    FinalizarConfiguração(C,C´)
    AtivarConfiguração(C´)
  Fim

Transações do 3º. Nível:
  TestarPerfil(Conf) : testa condições iniciais da configuração indicada

  SelecionarConfiguração() : retorna próxima configuração disponível no contrato

  AtivarConfiguração(Conf)
  Início
    Enquanto houver comandos na descrição da configuração indicada
      Obter próximo comando
      Configurador executa comando
      Colocar comando no Histórico de Operações Bem Sucedidas
    Fim-Enquanto
  Fim

  FinalizarConfiguração(Conf1,Conf2)
  Início
    Obter operações úteis a Conf2
    Enquanto houver comandos no Histórico de Operações Bem Sucedidas
              - Obter próximo comando
              - Se comando não for útil a Conf2,
                    então Configurador desfaz comando
    Fim-Enquanto
  Fim
```

Figura 3. Pseudocódigo das transações de 2º. e 3º. níveis do modelo proposto

Os dados do contrato de adaptação vigente servem de entrada para o gerenciamento da configuração. Dele são extraídos a lista de configurações disponíveis, com as instruções e condições para a implantação de cada uma delas. Tais dados são úteis na execução das transações *Selecionar Próxima Configuração* e *Testar Condições de Implantação*. As transações *Finalizar Configuração Corrente* e *Ativar Configuração Selecionada* envolvem a execução das instruções obtidas na descrição da configuração selecionada. Na finalização da configuração corrente, as operações básicas realizadas são desfeitas, mediante consulta ao histórico de operações realizadas com sucesso. Assume-se que a semântica da linguagem de descrição permita associar as instruções de configuração aos pares, com uma possuindo efeito contrário à outra, e que o configurador seja capaz de identificar isso. Por exemplo, no CR-RIO podem ser definidos os pares {`assert`; `remove`} e {`link`; `unlink`}. A utilização do histórico pode ser otimizada para que se desfaçam somente operações não necessárias à nova

configuração a ser implantada. Assim, são preservados componentes e ligações que serão úteis à nova configuração. Para auxiliar isso, define-se uma operação ***util*(C)**, que retorna a lista de operações necessárias à uma configuração C que já foram realizadas sem falha e estão presentes no histórico de operações concluídas.

5. Validação da proposta

Para validar a proposta deste trabalho, será descrito um exemplo aplicável ao *framework* CR-RIO. O exemplo consiste em uma aplicação cliente que utiliza um serviço oferecido por um provedor de informação (no exemplo, cotação da Bolsa de Valores), configurando-se uma arquitetura cliente-servidor típica. Para o exemplo, serão considerados dois provedores diferentes, e cada um deles pode oferecer dois tipos de serviço: com criptografia e sem criptografia. Cliente e provedor podem interagir através de conexão de rede fixa ou sem fio. Por questões de segurança, é desejo dos usuários da aplicação que a informação proveniente do serviço utilizado seja criptografada caso seja utilizada a conexão sem fio. Nessa situação, deve ser utilizado um componente para decodificar a informação antes que ela seja utilizada pelo cliente.

A Figura 4 ilustra o cenário obtido. Na Figura 4a, é mostrada a arquitetura genérica, com a aplicação cliente (AP) ligada ao provedor de serviço (PS) através de alguma conexão de rede (C). A Figura 4b ilustra as possíveis configurações para a aplicação, considerando-se as instâncias reais dos componentes disponíveis para configuração: os dois provedores (PS1 e PS2) e os tipos de serviço oferecidos (N para normal, K para criptografada), as conexões de rede (F para fixa, W para sem fio), e o componente decodificador (DK) no caso de utilização do serviço com criptografia.

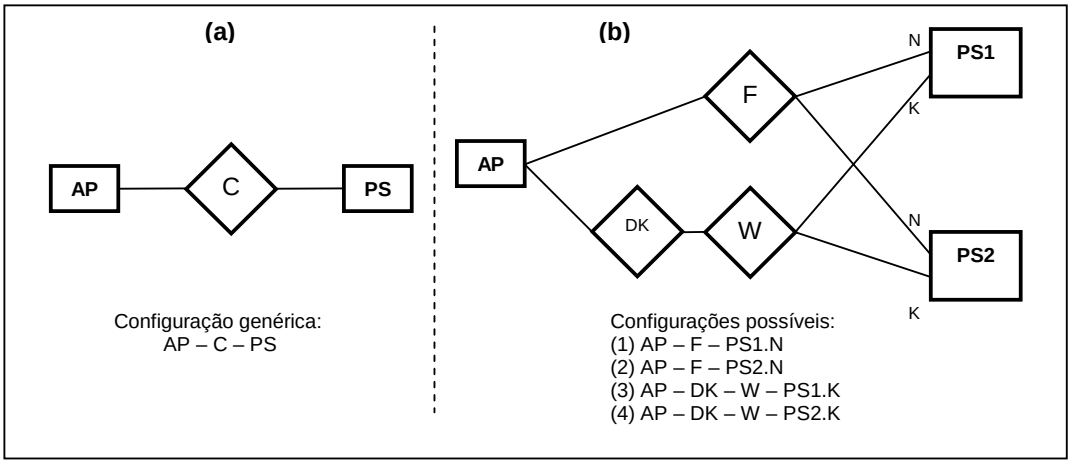


Figura 4. Exemplo de aplicação cliente-servidor: (a) Arquitetura genérica; (b) Configurações possíveis com componentes reais

5.1. Descrição no *framework* CR-RIO

O *framework* CR-RIO utiliza a linguagem CBabel para descrever arquiteturas e contratos. Para o exemplo proposto, são apresentados a seguir na Figura 5: as descrições da arquitetura básica da aplicação e os elementos do contrato para gerenciamento da configuração – configurações reais disponíveis e possibilidades de transição entre elas.

```
1.    module ClientServer {
2.      module Client { out port getQuote }
3.      module Provider { in port getQuoteN
4.                        in port getQuoteK }
5.      connector NetworkConnection: external;
6.      connector Decrypt: external;
7.      assert Client as AP;
8.      assert Provider as PS;
9.      assert NetworkConnection as C;
10.     link AP to PS by C
11.   }
12.   category ProviderNFP {
13.     responseTime: dynamic decreasing numeric ms; }
14.   category NetworkNFP {
15.     technology: static enum (fixed, wireless); }
16.   contract {
17.     service { assert Provider as PS1 at provider1.com monitoring ProviderNFP;
18.               assert Provider as PS2 at provider2.com monitoring ProviderNFP;
19.     } initProvider;
20.     negotiation { initProvider -> no-service }
21.   } startUpcontract;
22.   contract {
23.     _provider: Provider;
24.     profile { ProviderNFP.responseTime < 50; } providerSelection;
25.     profile { NetworkNFP.technology = fixed; } fNet;
26.     profile { NetworkNFP.technology = wireless; } wNet;
27.     service { assert Client as AP at <clientHost>;
28.               assert NetworkConnection as F with fNet;
29.               _provider = select Provider with providerSelection;
30.               link AP.getQuote to _provider.getQuoteN by F;
31.     } fixedNetwork;
32.     service { assert Client as AP at <clientHost>;
33.               assert NetworkConnection as W with wNet;
34.               assert Decrypt as DK at <clientHost>;
35.               _provider = select Provider with providerSelection;
36.               link AP.getQuote to _provider.getQuoteK by DK, W;
37.     } wirelessNetwork;
38.     negotiation { not fixedNetwork -> (wirelessNetwork -> no-service)
39.                   wirelessNetwork -> fixedNetwork
40.                   not wirelessNetwork -> no-service
41.     }
42.   } adaptationContract;
```

Figura 5. Descrições da arquitetura básica e contratos para o exemplo citado

Explicações sobre as descrições apresentadas:

- Descrição da Arquitetura Básica (linhas de 1 a 11): define os tipos de componente (módulos) utilizados: aplicacão cliente (`Client`, linha 2) e provedor (`Provider`, linhas 3-4). A porta `getQuote` indica a operação de solicitação da cotação da bolsa realizada pelo cliente, e as portas `getQuoteN` e `getQuoteK` indicam as operações disponibilizadas no provedor: sem criptografia e com criptografia. Os tipos de conectores `NetworkConnection` (conexão de rede, linha 5) e `Decrypt` (decriptação, linha 6) são declarados com externos, ou seja, providos pela plataforma de suporte à execução da aplicação. Nas linhas de 7 a 9 são descritas validações (`assert`) de instâncias genéricas dos componentes da aplicação (`AP`, `PS` e `C`), e na linha 10 é definida a restrição básica para composição da aplicação: o cliente liga-se ao provedor através de alguma conexão de rede disponível;

- Categorias (linhas de 12 a 15): indicam a lista de propriedades não-funcionais de interesse em um determinado recurso ou componente. A categoria `ProviderNFP` apresenta a propriedade tempo de resposta (`responseTime`) para um provedor de serviço. A declaração `dynamic` indica que a propriedade

possui valor dinâmico, e deverá ser monitorada em tempo de execução, e `decreasing` indica que menores valores são mais desejáveis na monitoração. A categoria `NetworkNFP` apresenta a propriedade tecnologia (`technology`) utilizada para uma conexão de rede, como sendo uma enumeração com dois valores possíveis: `fixed` e `wireless`. A propriedade é estática (`static`), o que indica que seu valor será atribuído e não sofrerá alteração.

- Contrato para validação dos provedores (linhas de 16 a 21): serve para adicionar referências dos provedores ao registro de componentes disponíveis, já que a inicialização deles é independente. O serviço `initProvider` apresenta as declarações para validação das duas instâncias, `PS1` e `PS2`, nos seus respectivos endereços, indicando a monitoração das propriedades definidas na categoria `ProviderNFP` – nesse caso, o tempo de resposta. No contrato, a cláusula `negotiation` representa uma máquina de estados. A configuração do serviço `initProvider` será tentada, e caso não seja bem sucedida, acontecerá uma transição para o estado de serviço indisponível (`no-service`), sinalizando que não foi possível a validação. Este contrato é independente do contrato de gerenciamento de configuração, e não será tratado no exemplo.
- Contrato de gerenciamento da configuração (linhas de 22 a 42): Nele, são definidos dois serviços, que descrevem as possibilidades de configuração: `fixedNetwork` (linhas 27-31) utiliza conexão de rede fixa, e `wirelessNetwork` (linhas 32-37) utiliza conexão sem fio com criptografia. Nos serviços, são indicadas as seguintes atividades:
    - Validação do cliente (linhas 27 e 32) e do conector de rede (linhas 28 e 33), de acordo com os respectivos perfis: tecnologia fixa (`fNet`, linha 25) e tecnologia sem fio (`wNet`, linha 26)
    - A seleção dinâmica (`select`) de uma instância de provedor (linhas 29 e 35) segundo o perfil `providerSelection` (linha 24), que define que o tempo de resposta monitorado deve ser inferior a 50ms. A instância selecionada é armazenada na variável `_provider`;
    - A ligação (`link`) entre o cliente e o provedor, utilizando os conectores adequados (linhas 30 e 36);
    - Diferenças entre os dois serviços: (i) utilização do conector `F` para rede fixa e `W` para sem fio; (ii) ligações entre as portas de cliente e provedor: sem criptografia (`getQuoteN`) no serviço de rede fixa, e com criptografia (`getQuoteK`) no serviço de rede sem fio; (iii) utilização do conector de decriptação (`DK`) no serviço de rede sem fio;
- A cláusula de negociação (linhas de 38 a 40) mostra que a configuração `fixedNetwork` é preferencial. Transições para `wirelessNetwork` e `no-service` não são automáticas, ou seja, somente acontecerão se a primeira não estiver disponível. Este fato, indicado pela declaração `not`, ocorre do mesmo modo na transição de `wirelessNetwork` para `no-service`. A transição de `wirelessNetwork` para `fixedNetwork` é automática – acontecerá caso a primeira esteja ativa e a segunda torne-se disponível.

5.2. Aplicação do contrato de adaptação

São apresentadas quatro situações para ilustrar a aplicação do contrato de gerenciamento de configuração descrito para o exemplo. O objetivo é demonstrar o tratamento de falhas na adaptação, e também a utilização otimizada do histórico de operações concluídas. Será assumido que as referências dos provedores de serviço já estão devidamente adicionadas ao registro de componentes. Nas descrições das situações, **C** representa a configuração corrente, e **C'** representa uma nova configuração selecionada.

As situações acontecem na seguinte ordem:

1º) Imposição da configuração `fixedNetwork` como primeira configuração;
2º) Mudança no provedor, mantendo-se a conexão de rede fixa;
3º) Mudança no provedor, porém com falha na conexão de rede fixa;
4º) Mudança de conexão de rede, de sem fio para fixa.

Para cada situação, são mostrados: (i) a informação de contexto com as propriedades monitoradas; (ii) a execução do algoritmo mostrado na Seção 4.2; (iii) o estado do histórico de operações concluídas, atualizado a cada operação executada; (iv) a ocorrência de falhas de configuração e respectivo tratamento. Para simplificar, a operação `link` foi resumida, omitindo-se as portas envolvidas na ligação.

Situação 1: Imposição da primeira configuração

Contexto:

Configuração corrente: C = *null* (nenhuma configuração ativa)

Propriedades: `PS1.ProviderNFP.responseTime = 30` ←
`PS2.ProviderNFP.responseTime = 35`

Execução da transação *Implantar Nova Configuração*:

1. *Selecionar próxima configuração* (selecionada: C' = `fixedNetwork`)
2. *Testar condições iniciais* (perfil `fNet`: OK)
3. *Finalizar configuração corrente*: *nada a fazer*
4. *Ativar configuração selecionada*:

`assert Client as AP at <clientHost>` → OK (cliente validado)
histórico: `OP1 - assert Client as AP at <clientHost>`

`assert NetworkConnection as F with fNet` → OK (conexão validada)
histórico:
```
OP1 - assert Client as AP at <clientHost>
OP2 - assert NetworkConnection as F with fNet
```

```
_provider = select Provider with providerSelection
```
instância selecionada: `PS1` (menor tempo de resposta)

`link AP to _provider by F` : OK (ligação criada: `AP–F–PS1`)
histórico:
```
OP1 - assert Client as AP at <clientHost>
OP2 - assert NetworkConnection as F with fNet
OP3 - link AP.getQuote to PS1.getQuoteN by F
```

Conclusão: Configuração `fixedNetwork` ativada com sucesso(`AP–F–PS1`)

Situação 2: Mudança de provedor (seleção dinâmica)

Contexto:

Configuração corrente: C = `fixedNetwork (AP–F–PS1)`

Propriedades: `PS1.ProviderNFP.responseTime = 30`
`PS2.ProviderNFP.responseTime = 25` ←

Histórico:
```
OP1 - assert Client as AP at <clientHost>
OP2 - assert NetworkConnection as F with fNet
OP3 - link AP.getQuote to PS1.getQuoteN by F
```

Execução da transação *Implantar Nova Configuração*:

1. *Selecionar próxima configuração* (selecionada: C' = `fixedNetwork`)
2. *Testar condições iniciais* (perfil `fNet`: OK)
3. *Finalizar configuração corrente*

   Verificar operações úteis no histórico: *util* (C') = { `OP1`, `OP2` }

   Finalização: Desfazer `OP3` :`unlink AP from PS1`

   Não desfazer `OP2`

   Não desfazer `OP1`
4. *Ativar configuração selecionada*:

   `assert Client as AP at <clientHost>` → OK (aproveitada)

   histórico: `OP1 - assert Client as AP at <clientHost>`

   `assert NetworkConnection as F with fNet` → OK (aproveitada)

   histórico:
   ```
   OP1 - assert Client as AP at <clientHost>
   OP2 - assert NetworkConnection as F with fNet
   ```
   ```
   _provider = select Provider with providerSelection
   ```

   instância selecionada: `PS2` (menor tempo de resposta)

   `link AP to _provider by F` : OK (ligação criada: `AP–F–PS2`)

   histórico:
   ```
   OP1 - assert Client as AP at <clientHost>
   OP2 - assert NetworkConnection as F with fNet
   OP3 - link AP.getQuote to PS2.getQuoteN by F
   ```

Conclusão: Configuração `fixedNetwork` ativada com sucesso (`AP–F–PS2`)

Situação 3: Mudança de provedor com falha na conexão após teste inicial

Contexto:

Configuração corrente: C = `fixedNetwork (AP–F–PS2)`

Propriedades:
```
PS1.ProviderNFP.responseTime = 20 ←
PS2.ProviderNFP.responseTime = 25
```

Histórico:
```
OP1 - assert Client as AP at <clientHost>
OP2 - assert NetworkConnection as F with fNet
OP3 - link AP.getQuotes to PS2.getQuoteN by F
```

Execução da transação *Implantar Nova Configuração*:

1. *Selecionar próxima configuração* (selecionada: C' = `fixedNetwork`)
2. *Testar condições iniciais* (perfil `fNet`: OK)
3. *Finalizar configuração corrente*

   Verificar operações úteis no histórico: *util* (C') = { `OP1`, `OP2` }

   Finalização: Desfazer `OP3` :`unlink AP from PS2`

   Não desfazer `OP2`

   Não desfazer `OP1`
4. *Ativar configuração selecionada*:

   `assert Client as AP at <clientHost>` → OK (aproveitada)

   histórico: `OP1 - assert Client as AP at <clientHost>`

   `assert NetworkConnection as F with fNet` → FALHOU!

   Tratamento: Abortar transação corrente

   Iniciar nova transação *Implantar Nova Configuração*

Execução da nova transação *Implantar Nova Configuração*:

1. *Selecionar próxima configuração* (selecionada: C' = `wirelessNetwork`)
2. *Testar condições iniciais* (perfil `wNet`: OK)
3. *Finalizar configuração corrente*

   Verificar operações úteis no histórico: *util* (C') = { `OP1` }

   Finalização: Não desfazer `OP1`
4. *Ativar configuração selecionada*:

   `assert Client as AP at <clientHost>` → OK (aproveitada)

   histórico: `OP1 - assert Client as AP at <clientHost>`

   `assert NetworkConnection as W with wNet` → OK (conexão validada)

   histórico:
   ```
   OP1 - assert Client as AP at <clientHost>
   OP2 - assert NetworkConnection as W with wNet
   ```
   `assert Decrypt as DK at <clientHost>` → OK (conector validado)

   histórico:
   ```
   OP1 - assert Client as AP at <clientHost>
   OP2 - assert NetworkConnection as W with wNet
   OP3 - assert Decrypt as DK at <clientHost>
   ```
   `_provider = select Provider with providerSelection`

   instância selecionada: `PS1` (menor tempo de resposta)

   `link AP to _provider by DK, W` : OK (ligação criada: `AP–DK-W–PS1`)

   histórico:
   ```
   OP1 - assert Client as AP at <clientHost>
   OP2 - assert NetworkConnection as W with wNet
   OP3 - assert Decrypt as DK at <clientHost>
   OP4 - link AP.getQuote to PS1.getQuoteK by DK, W
   ```

Conclusão: Configuração `wirelessNetwork` ativada com sucesso (`AP–DK-W–PS1`)

Situação 4: Mudança de conexão de rede (conexão fixa disponível)

Contexto:

Configuração corrente: C = `wirelessNetwork (AP–DK-W–PS1)`

Propriedades: `PS1.ProviderNFP.responseTime = 20 ←`
`PS2.ProviderNFP.responseTime = 25`

Histórico:
```
OP1 - assert Client as AP at <clientHost>
OP2 - assert NetworkConnection as W with wNet
OP3 - assert Decrypt as DK at <clientHost>
OP4 - link AP.getQuote to PS1.getQuoteK by DK, W
```

Execução da transação *Implantar Nova Configuração*:

1. *Selecionar próxima configuração* (selecionada: C' = `fixedNetwork`)
2. *Testar condições iniciais* (perfil `fNet`: OK)
3. *Finalizar configuração corrente*

   Verificar operações úteis no histórico: *util* (C') = { `OP1` }

   Finalização: Desfazer `OP4` : `unlink AP from PS1`

   Desfazer `OP3` : `remove DK from <clientHost>`

   Desfazer `OP2` : `remove W`

   Não desfazer `OP1`
4. *Ativar configuração selecionada*:

   `assert Client as AP at <clientHost>` → OK (aproveitada)

   histórico: `OP1 - assert Client as AP at <clientHost>`

   `assert NetworkConnection as F with fNet` → OK (conexão validada)

   histórico:
   ```
   OP1 - assert Client as AP at <clientHost>
   OP2 - assert NetworkConnection as F with fNet
   ```
   `_provider = select Provider with providerSelection`

   instância selecionada: `PS1` (menor tempo de resposta)

`link AP to _provider by F` : OK (ligação criada: `AP–F–PS1`)

histórico:
```
OP1 - assert Client as AP at <clientHost>
OP2 - assert NetworkConnection as F with fNet
OP3 - link AP.getQuote to PS1.getQuoteN by F
```

Conclusão: Configuração `fixedNetwork` ativada com sucesso (`AP–F–PS1`)

6. Análise do exemplo e conclusões

As situações descritas na Seção 5 cobrem as possibilidades para adaptação citadas na Seção 4.1: primeira configuração (Situação 1), falha em componentes (Situação 3) e invalidação de requisitos de qualidade (Situações 2 e 4). O agrupamento de transações em níveis permite interromper uma transação mais interna e tratar a interrupção – no caso, uma exceção de software – sem prejuízo para a transação que a envolve. Assim, é possível manter a propriedade de atomicidade da transação mais externa.

Com a captura de falhas a cada execução de uma operação básica, é possível interromper a transação de implantação de uma configuração (2º. Nível), e desfazer as operações já realizadas, caso não sejam úteis à nova configuração. O histórico de operações concluídas serve a esse propósito, funcionando como uma pilha. Como uma nova transação de 2º. Nível é iniciada a cada falha, garante-se que a transação mais externa (1º. Nível) só poderá terminar de duas maneiras: ou uma nova configuração é implantada com sucesso, ou é sinalizada a impossibilidade de implantar qualquer configuração, com a ocorrência da Exceção 1 (Nenhuma Configuração Disponível). Assim a atomicidade da adaptação é garantida, e logo, a manutenção da consistência.

O modelo de adaptação proposto é aplicável a qualquer tipo de arquitetura que possa ser descrita segundo o modelo de componentes de CR-RIO. Incluem-se aí os estilos arquiteturais mais comuns, como o cliente-servidor ilustrado no exemplo, e suas variações como P2P, em que componentes são clientes e servidores ao mesmo tempo,e *pipeline*, em que a saída de um componente é ligada à entrada de outro. O ponto chave é ter disponíveis as descrições arquiteturais e contratos de qualidade de serviço. É possível a utilização de contratos dinâmicos, que se apóiam em serviços de descoberta de componentes e técnicas de otimização e inteligência computacional, por exemplo.

Dois pontos importantes não foram cobertos pelo escopo deste artigo: a possibilidade de execução concorrente dos passos da adaptação, e a possibilidade de falha e reinício do gerenciador de configuração. Para fins de simplificação adotou-se a execução sequencial da adaptação e assumiu-se a robustez do gerenciador de configuração. Tais pontos, porém, merecem destaque em um trabalho mais amplo.

Direções futuras para o trabalho apontam para a aplicação em uma extensão do *framework* CR-RIO para configuração de serviços Web [W3C 2008] e tecnologias associadas: descrição de serviços com WSDL (*Web Service Description Language*), descoberta de recursos com UDDI (*Universal Description Discovery and Integration*) e interação entre componentes através de SOAP (*Simple Object Access Protocol*).

7. Referências

Batista, T., Joolia, A. e Coulson, G. (2005). “Managing Dynamic Reconfiguration in Component-Based Systems”. *Lecture Notes in Computer Science, Vol. 3527/2005*, Springer.

Beugnard, A., Jézéquel, J.-M., Plouzeau, N. e Watkins, D. (1999). “Making Components Contract Aware”. *IEEE Computer, 32(7):38-45*.

Capra, L., Zachariadis, L. e Mascolo, C. (2005). “Q-CAD: QoS Context-Aware Discovery Framework for Adaptive Mobile Systems”. Em *IEEE International Conference on Pervasive Services 2005*. Santorini (Grécia), julho.

Cardoso, L., Sztajnberg, A. e Loques, O. (2006). “Self-adaptive Applications Using ADL Contracts”. Em *SelfMan 2006 – 2 nd . IEEE International Workshop on Self-Managed Networks, Systems and Services*. Dublin (Irlanda), junho.

Garlan, D., Cheng, S., Huang, A., Schmerl, B. e Steenkiste, P. (2004) “Rainbow: Architecture Based Self-Adaptation with Reusable Infrastructure”. *IEEE Computer, 18(10): 46-54*.

Gray, J. e Renter, A. (1993). *Transaction Processing: Concepts and Techniques*. Kaufman Publishers.

Kephart, J. O. e Chess, D. M. (2003). “The Vision of Autonomic Computing”. *IEEE Computer, 36(1):41-50*.

Kon, F. e Campbell, R. H. (2000). “Dependence Management in Component-Based Distributed Systems”. *IEEE Concurrency, 8(1):26-36*.

Kramer, J. e Magee. J. (1985). “Dynamic Configuration for Distributed Systems”, *IEEE Transactions on Software Engineering, SE-11(4):424-436*.

Loyall, J. P., Schantz, R. E., Zinky, J. A. e Bakken, D. E. (1998). “Specifying and Measuring Quality of Service in Distributed Object Systems”. Em *The 1st IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*. Kyoto (Japão), abril.

Lynch, N. A., Merrit, M., Weihl, W. E. e Fekcte, A. (1993). *Atomic Transactions*, Morgan Kaufman.

Meyer, B. (1992). “Appliying ‘Design by Contract’”. *IEEE Computer, 25(10):40-51*.

Microsoft Corporation (acessado em abril de 2008). *Microsoft's Developer News (MSDN) Library*. http://msdn.microsoft.com/library/default.asp

OMG - Object Management Group (acessado em abril de 2008). *Catalog of OMG Specifications*. http://www.omg.org/technology/documents/spec_catalog.htm

Romanovsky, A. (2001). “Coordinated Atomic Actions: How To Remain ACID in the Modern World”, *Software Engineering Notes 26(2):66-68*, ACM SIGSOFT.

Shaw, M. e Garlan, D. (1996). *Software Architecture: Perspectives on an Emerging Discipline*, 1 st . ed., Prentice Hall.

Sun Microsystems (acessado em abril de 2008). *Developer Services: Information about Java technology*. http://developer.java.sun.com

W3C - World Wide Web Consortium (acessado em abril de 2008). *Web Services Architecture.* http://www.w3.org/TR/ws-arch/

Simulação de um Algoritmo de Diagnóstico Distribuído para Redes Particionáveis de Topologia Arbitrária

Andréa Weber[1,2], Aline Wolpert dos Santos[1],
Elias Procópio Duarte Jr.[1], Keiko V. O. Fonseca[2]

[1]Departamento de Informática – Universidade Federal do Paraná (UFPR)
Caixa Postal 19081 – 81531-990 – Curitiba – PR – Brazil

[2]Prog. de Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI)
Universidade Tecnológica Federal do Paraná (UTFPR)
Curitiba – PR - Brazil

{andrea,aws03,elias}@inf.ufpr.br, keiko@utfpr.edu.br

Abstract. *This work presents experimental results of the evaluation of the* Distributed Network Reachability *algorithm. DNR is a distributed system-level diagnosis algorithm that allows every node of a partitionable general topology network to determine which portions of the network are reachable and unreachable. Extensive simulation results of dynamic fault and repair events on nodes and links are presented. The systems considered are highly dynamic, the occurrence of the events is modeled by a Poisson process. All the results are presented within a* 95% *confidence interval. Simulated topologies included random graphs as well as regular graphs, such as meshes and hypercubes.*

Resumo. *Este trabalho apresenta resultados experimentais da avaliação do algoritmo* Distributed Network Reachability. *DNR é um algoritmo de diagnóstico distribuído em nível de sistema que permite que cada nodo de uma rede particionável de topologia arbitrária determine quais porções da mesma estão atingíveis e inatingíveis. São apresentados extensivos resultados de simulação de eventos dinâmicos de falha e recuperação de nodos e enlaces. São considerados sistemas fortemente dinâmicos, com a ocorrência de eventos modelada por um processo de Poisson. Todos os resultados apresentados têm intervalo de confiança de* 95%*. Os resultados foram obtidos para grafos aleatórios e também para topologias regulares, como* meshes *e hipercubos.*

1. Introdução

Sistemas computacionais de grande porte baseados em múltiplos processadores, tais como computadores paralelos, *clusters* e redes de computadores são plataformas críticas para diversos tipos de organizações. É altamente desejável que tais sistemas se mantenham operando apesar da falha de alguns de seus componentes. Quando partes do sistema se tornam desconectadas, atividades de reconfiguração devem ser executas com presteza. Neste trabalho são apresentados resultados de uma avaliação experimental do algoritmo *Distributed Network Reachability* (DNR) para monitoramento *on line* tolerante a falhas de sistemas particionáveis baseados em redes de topologia arbitrária. O algoritmo pode ser utilizado para construir um mapa refletindo quais porções da rede estão atingíveis e inatingíveis do ponto de vista de qualquer nodo da rede.

O algoritmo *Distributed Network Reachability* (DNR) [Weber 2008] é o primeiro algoritmo distribuído de diagnóstico em nível de sistema para redes de topologia arbitrária que executa na presença de particionamentos e reconexões da rede causados por eventos dinâmicos de falha e recuperação de nodos e enlaces. O algoritmo consiste de três fases: teste, disseminação e cálculo de alcançabilidade. Durante a fase de testes, cada enlace é testado por um de seus nodos adjacentes em intervalos de teste alternados. Quando da detecção de um novo evento, i.e., um enlace *sem-falha* se torna *não-respondendo* ou vice-versa, o testador inicia a fase de disseminação, na qual uma estratégia paralela é utilizada para informar os outros nodos alcançáveis sobre o evento. Novos eventos podem ocorrer e ser detectados antes que a disseminação seja completada. Sempre que um evento é detectado ou informado, a terceira fase é executada, na qual um algoritmo de conectividade em grafos é empregado para computar a alcançabilidade da rede.

O desempenho do algoritmo foi avaliado através de simulação, considerando múltiplos eventos de falha e recuperação, tanto de nodos como de enlaces. Além disso, tanto as situações na qual a rede sofre particionamento como na qual ela permanece sempre conectada foram avaliadas. Experimentos foram executados tanto em topologias aleatórias, como em topologias regulares como *meshes* e hipercubos. O programa de simulação foi construído usando a biblioteca de simulação de eventos discretos SMPL [MacDougall 1987]. Os experimentos foram realizados no *cluster* de alto desempenho da Universidade Federal do Paraná, composto por cerca de 60 máquinas multiprocessadas conectadas em rede *Myrinet* e *Gigabit-Ethernet*.

O restante do artigo é organizado como segue. A seção 2 apresenta trabalhos relacionados. Na seção 3 o algoritmo DNR é descrito. A seção 4 apresenta resultados de simulação da ocorrência de um grande número de eventos concorrentes em várias topologias aleatórias e regulares. A conclusão segue na seção 5.

2. Trabalhos Relacionados

Muitos algoritmos de diagnóstico em nível de sistema para redes de topologia arbitrária tem sido propostos. [Bagchi and Hakimi 1991] introduziram o primeiro algoritmo de diagnóstico para este tipo de rede, o qual era executado *off-line* a partir de resultados de testes prévios. [Stahl, Buskens and Bianchini 1992] introduziram e avaliaram através de simulação o algoritmo Adapt, que pode ser executado *on line*: quando um nodo falha, os outros nodos reconectam o grafo de testes. A estratégia de disseminação de eventos empregada é sequencial. [Rangarajan, Dahbura and Ziegler 1995] introduziram o algoritmo RDZ para diagnóstico em nível de sistema de redes de topologia arbitrária. O algoritmo constrói um grafo de testes que garante um número ótimo de testes, e utiliza uma estratégia paralela de disseminação baseada em inundação. O algoritmo, entretanto, não garante o diagnóstico de algumas configurações de falhas.

[Duarte et. al. 1997] introduziram um algoritmo que utiliza uma estratégia de testes chamada *two-way testing*, em que o nodo testado determina o estado do testador. Uma estratégia de testes baseada em *token* foi proposta posteriormente, na qual os nodos conectados por um enlace se alternam nos papéis de testado e testador. Informações sobre eventos detectados são disseminadas utilizando uma árvore distribuída de busca em largura. O algoritmo DNC (*Distributed Network Connectivity*) [Duarte Jr. and Weber 2003] baseia-se naquele algoritmo, mas suporta eventos dinâmicos, i.e. durante a fase de disseminação

novos eventos podem ocorrer e a disseminação de todos é garantida. Posteriormente uma outra estratégia de testes foi desenvolvida [Weber, Duarte Jr. and Fonseca 2006] a qual garante a resolução de casos que levam a testes ditos simultâneos.

[Subbiah and Blough 2004] introduziram o algoritmo ForwardHeartbeat, que é baseado em *heartbeats* e permite o diagnóstico em redes de topologia arbitrária numa situação dinâmica de ocorrência de eventos. O algoritmo não permite particionamento da rede. [Khanna et. al. 2007] apresentam uma aplicação que realiza diagnóstico de falhas em protocolos de redes de larga escala. A aplicação considera as entidades a serem diagnosticadas como caixas pretas e observa a troca de mensagens entre elas. Um grafo causal é construído para seguir a propagação do erro e identificar sua fonte.

Algoritmos de diagnóstico distribuído para redes sem-fio também são trabalhos relacionados. As topologias das redes sem-fio não são apenas arbitrárias como também dinâmicas. [Santi and Blough 2002] consideram o problema de calcular o quão grande o raio de transmissão dos nodos em uma rede móvel *ad hoc* deve ser de forma a assegurar que a rede resultante seja conectada; eles também computam o raio para assegurar a conectividade durante uma dada fração de tempo quando os nodos são móveis. [Chessa and Santi 2002] propuseram um protoloco distribuído para diagnóstico de falhas em redes de sensores. O protocolo é iniciado por um observador externo e tanto o diagnóstico como a disseminação de seu resultado ocorrem ao longo de uma árvore. [Ding et. al. 2005] apresentam um algoritmo tolerante a falhas para detectar nodos falhos e a borda de propagação de um evento em uma rede de sensores. [Elhadef, Boukerche and Elkadiki 2007] propõem um algoritmo adaptativo de diagnóstico distribuído baseado em comparações para redes móveis *ad hoc*, baseado no trabalho de [Chessa and Santi 2001]. O resultado dos testes realizados localmente pelos sensores é disseminado pela rede por meio de um árvore geradora mínima adaptável. [Lee, M. H. and Choi Y. H. 2007] apresentam um algoritmo distribuído eficiente baseado no modelo de comparações para isolar nodos falhos em uma rede de sensores, utilizando a premissa de que nodos sem-falha apresentam valores de leitura similares aos de seus vizinhos.

3. Descrição do Algoritmo

Nesta seção é descrito o algoritmo *Distributed Network Reachability* [Weber 2008] para redes particionáveis de topologia arbitrária. Um nodo executando o algoritmo obtém informação sobre o componente conexo da rede ao qual pertence, i.e., quais porções da rede estão alcançáveis e inalcançáveis.

Numa rede de topologia arbitrária não há necessariamente um canal de comunicação entre quaisquer pares de nodos. Tanto nodos como enlaces podem se tornar falhos, considerando falhas *crash* que podem ser reparadas. Uma falha pode particionar a rede, que pode subsequentemente reconectar. Considerando um par de nodos conectado por um enlace, se os testes executados em um nodo pelo outro determinam que o enlace não está respondendo, é impossível distinguir se é o nodo testado ou o enlace que está falho. Desta forma, falhas em redes de topologia arbitrária são ditas ambíguas [Duarte et. al. 1997]. Além disso, como o sistema não é síncrono, um *timeout* pode ser causado por um nodo falho ou muito lento. Neste caso, quando não há resposta a um teste, o enlace é considerado *não-respondendo*. A qualquer instante de tempo, um nodo

não falho executando DNR classifica os outros nodos em um de dois estados: *sem-falha* ou *inatingível*. Um enlace no qual um nodo sem-falha está conectado pode estar *sem-falha* ou *não-respondendo*, e outros enlaces podem estar *sem-falha*, *não-respondendo* ou *inatingíveis*. Um nodo sem-falha considera um enlace como *inatingível* quando o enlace é não adjacente a nenhum outro nodo alcançável sem-falha.

O algoritmo consiste de três fases: teste dos enlaces locais, disseminação de informação sobre novos eventos e cálculo local de alcançabilidade. Testes são executados periodicamente em intervalos de teste, que são intervalos de tempo. O assinalamento de testes é dinâmico, de forma que dois nodos que compartilham um enlace de comunicação se alternam nos papéis de *testador* e *testado* e um número ótimo de testes por enlace por intervalo de testes é assegurado: um teste por enlace por intervalo de testes, desde que ambos os nodos adjacentes estejam sem-falha. O algoritmo emprega uma estratégia paralela de disseminação que apresenta uma latência de disseminação proporcional ao diâmetro da rede. Ambos nodos e enlaces podem falhar e recuperar durante a execução do algoritmo. A rede pode particionar e subsequentemente reconectar.

Sob o ponto de vista de um nodo testador, um *evento* é definido como uma mudança de estado de um enlace, ou de *sem-falha* para *não-respondendo* ou de *não-respondendo* para *sem-falha*. Embora um nodo execute testes enviando uma requisição de testes para um nodo vizinho, um evento é sempre descrito em termos do enlace de comunicação correspondente. Os nodos mantém uma visão local da topologia da rede representada por um grafo, no qual um *timestamp* é mantido para cada enlace. Os *timestamps* empregados são contadores de eventos similares àqueles propostos em [Rangarajan, Dahbura and Ziegler 1995]. Os *timestamps* para todos os enlaces são inicialmente 1 e todos os enlaces são assumidos como não-respondendo. A cada vez que um novo evento é detectado, o *timestamp* para aquele enlace é incrementado. Portanto, um *timestamp* par corresponde a um enlace *sem-falha*; um *timestamp* ímpar corresponde a um enlace *não-respondendo*.

O intervalo de testes leva em conta a tecnologia e os requisitos e é estimado de forma a evitar a perda de eventos. Ao iniciar, um nodo executando o algoritmo testa todos os seus vizinhos após expirar um intervalo de tempo pré-definido chamado *recovery wait time* [Subbiah and Blough 2004]. Durante este intervalo de tempo o nodo que está recuperando não envia requisições ou respostas a testes e não processa mensagens de disseminação. Isto permite que um nodo fique falho durante um tempo mínimo e garante que enlaces *não-respondendo* sejam sempre detectados. Levando em conta o *recovery wait time* e o intervalo de testes, são formalmente calculados os intervalos de tempo que nodos e enlaces devem se manter nos estados falho e sem-falha de forma que uma mudança de estados seja sempre detectada. Estes intervalos de tempo são designados como *state holding times* [Subbiah and Blough 2004].

O procedimento de teste adotado pelo algoritmo é dito *two-way* no sentido em que permite que ambos os nodos testador e testado descubram mutuamente se estão sem-falha. O assinalamento de testes é baseado em *token* e se apóia no teste *two-way*. Cada par de nodos conectado por um enlace mantém um *token*; o nodo que tem o *token* no início de um intervalo de testes é o testador, o outro é o nodo testado. Se ambos os nodos estão sem-falha durante o teste o *token* é transferido. O processo é repetido e garante que somente um teste seja executado por enlace por intervalo de testes.

No caso em que um nodo se torna falho, este pode ser tanto o nodo testado ou o testador para o próximo intervalo de testes. Se o nodo testado falha, o testador se manterá testando a cada dois intervalos de teste. Se o testador para o próximo intervalo de testes falha, o *token* desaparece. O nodo testado detecta que não recebeu uma requisição de teste e no intervalo de testes seguinte um *token* é criado. O nodo assume o papel de testador, executando um teste a cada dois intervalos de teste. No caso de o enlace se tornar falho, ambos os nodos testam a cada dois intervalos de teste, alternadamente. Após a recuperação do enlace, o primeiro teste feito por um dos nodos faz com que o outro responda e se torne o testador para o intervalo seguinte.

Quando um nodo recupera, ele cria um *token* para si mesmo e, após expirado o *recovery wait time*, testa todos os nodos ao qual está conectado. Testes ditos *simultâneos* podem ocorrer se ambos os nodos recuperam praticamente ao mesmo tempo e se tornam testadores, ou quando um nodo recupera e testa um vizinho exatamente ao mesmo tempo em que é testado por ele. Quando ambos os nodos recebem, um do outro, uma requisição de testes, a situação é detectada e um critério baseado no identificador dos nodos é utilizado para determinar qual nodo responderá ao teste, tornando-se o nodo testado.

A fase de disseminação é iniciada com a detecção de um novo evento. A estratégia de disseminação empregada é paralela, baseada em inundação, iniciada pelo testador que detectou o novo evento. Este nodo monta uma mensagem de disseminação contendo informação de diagnóstico. Cada mensagem de diagnóstico tem: (1) o identificador do nodo testador, (2) o identificador do vizinho testado, e (3) o *timestamp* do enlace testado. O nodo que inicia a disseminação envia a mensagem correspondente em todos os seus enlaces. Um nodo que recebe uma mensagem de disseminação verifica se ela contém ou não nova informação. A informação é nova quando contém *timestamps* maiores para um ou mais enlaces que aqueles contidos na tabela de enlaces local. Se a mensagem contém nova informação, o nodo envia esta informaÃ§Ã£o em todos os seus enlaces, exceto no enlace por onde a mensagem foi recebida, e desta forma sucessivamente até que a disseminação alcance todos os nodos.

Se nodos ou enlaces falham, a rede pode se particionar em dois ou mais componentes conexos. Ambos os nodos conectados pelo enlace detectam o evento e disseminam informação sobre o mesmo a todos os nodos sem-falha alcançáveis em cada partição. Um evento de *healing* pode ocorrer e é definido por um enlace ou nodo falhos que é reparado e pode fazer com que duas ou mais partições se reconectem em um único componente conexo. Quando um nodo recupera, eventos de *healing* ocorrem em todos os seus enlaces sem-falha. Um evento de *healing* é detectado pela chegada de uma requisição de teste sobre um enlace considerado *não-respondendo*. Ao tempo em que o nodo testado responde à requisição de teste, ele também percebe que o enlace está sem-falha, com os testes *two-way*. O nodo testado responde com uma mensagem de *healing*, a qual leva informação sobre os *timestamps* dos enlaces localizados em sua partição. Quando da recepção da mensagem de *healing*, o testador atualiza sua tabela de enlaces local, incrementa o *timestamp* do enlace recuperado e inicia a disseminação de informação sobre toda a tabela. Esta estratégia permite que ambos os nodos troquem informação sobre os componentes aos quais pertenciam, de forma que *timestamps* atualizados de enlaces previamente inalcançáveis atinjam toda a rede.

A cada vez que um nodo detecta ou recebe informação sobre um novo evento,

a terceira fase é executada, na qual um algoritmo de conectividade em grafos mostra a alcançabilidade da rede do ponto de vista do nodo executando o algoritmo.

4. Resultados Experimentais

Nesta seção são apresentados resultados de simulação do algoritmo DNR. O desempenho do algoritmo foi avaliado tanto em eventos de falha como em eventos de recuperação, nos casos em que tanto nodos como enlaces se tornaram falhos ou recuperaram. Além disso, tanto as situações na qual a rede sofre particionamento como na qual ela permanece sempre conectada foram avaliadas. Experimentos foram executados tanto em topologias aleatórias como em topologias regulares como *meshes* e hipercubos.

O programa de simulação foi construído usando a biblioteca de simulação de eventos discretos SMPL [MacDougall 1987]. Os experimentos foram realizados no *cluster* de alto desempenho da Universidade Federal do Paraná, composto por cerca de 60 máquinas, as principais com 16 processadores e 16 GB de memória cada, bem como máquinas com 8 processadores e 8 GB de memória cada. Os processadores incluem *Opteron* 1.8 GHz com 1MB de *cache* L2, Intel *quad-core* 2.4 GHz e Athlon, conectados por rede *Myrinet* e *Gigabit-Ethernet*. O ambiente nas máquinas é exclusivamente de 64 bits, baseado no sistema operacional Linux. Há um servidor central de disco no qual as máquinas de cálculo fazem *boot* remoto.

Os experimentos foram conduzidos de forma a medir as latências de diagnóstico de eventos de falha e de recuperação tanto em nodos como em enlaces. A latência de diagnóstico inclui a latência de detecção do evento, i.e., o tempo para um determinado nodo detectar o evento em um enlace adjacente, e a latência de disseminação de informação sobre o mesmo, i.e., o tempo necessário para que os demais nodos alcançáveis sem-falha tomem conhecimento do evento ocorrido. De forma a obter um intervalo de confiança igual a 95% para as médias das latências, cinco rodadas independentes foram inicialmente executadas para um dado tipo e tamanho de topologia; 5000 eventos de falha e recuperação (2500 cada) foram escalonados em cada rodada. A semi-amplitude do intervalo de confiança foi avaliada até ser menor ou igual a 10% da média obtida para todas as rodadas. O dinamismo da ocorrência de eventos foi dado por um processo de Poisson. Tão logo um evento ocorre em um nodo ou em um enlace, o evento seguinte no mesmo nodo ou enlace é escalonado para ocorrer após o *state holding time* correspondente mais um período de tempo exponencialmente distribuído. Os eventos são concorrentes, escalonados para ocorrer em qualquer nodo ou enlace. Duas médias para o processo de Poisson foram utilizadas: 1 segundo e 200 segundos, portanto foram simulados sistemas altamente dinâmicos. A menor média da distribuição exponencial corresponde a um ambiente mais dinâmico, onde nodos mudam de estado com mais frequência, pouco tempo depois de permanecer em um estado durante o *state holding time* correspondente. A média de 200 segundos para a distribuição exponencial simula o ambiente oposto, no qual os nodos permanecem em cada estado por mais tempo.

Um conjunto de experimentos foi conduzido de forma a comparar o desempenho dos algoritmos DNR e *ForwardHeartbeat*. Nestes experimentos, os parâmetros de simulação e o dinamismo da ocorrência de eventos são similares àqueles utilizados em [Subbiah and Blough 2004].

Os parâmetros de simulação estão listados na Tabela 1. Os parâmetors Δ_{send_init},

Δ_{send_min} e Δ_{send_max} significam, respectivamente, o tempo para um nodo iniciar uma comunicação e os tempos mínimo e máximo de atraso num canal de comunicação. O intervalo de testes é dado por π. Seus valores foram mantidos fixos em todos os experimentos.

Tabela 1. Parâmetros de Simulação

Parâmetros de Simulação	
Simulador	SMPL
Δ_{send_init}	0.002 segundos
Δ_{send_min}	0.008 segundos
Δ_{send_max}	0.08 segundos
π	30 segundos
Médias para o processo de Poisson	1 segundo, 200 segundos
# de rodadas iniciais	5
# de eventos por rodada	2500
Intervalo de confiança	95%

Para cada uma das topologias utilizadas nos experimentos, suas conectividades de vértices e de arestas foram avaliadas utilizando o algoritmo de Ford-Fulkerson [Cormen et. al. 2001]. A conectividade de vértices (arestas) de um grafo é o menor número de vértices (arestas) cuja remoção pode desconectar o grafo. Esta propriedade é utilizada para construir dois cenários de simulação: o cenário no qual particionamentos da rede não ocorrem e o cenário no qual a rede pode se tornar particionada. Ao simular eventos em nodos no primeiro cenário, o número máximo de $k-1$ nodos foi permitido estar no estado falho simultaneamente, tendo a rede conectividade de vértices igual a k. Similarmente para eventos em enlaces. Neste cenário, se o número de nodos ou enlaces falhos é igual a $k-1$ quando um evento de falha em nodo ou enlace vai ocorrer, um período de tempo exponencialmente distribuído é utilizado para reescalonar aquele evento para um instante de tempo posterior. Por outro lado, ao simular o cenário no qual a rede pode sofrer particionamento e reconexão, eventos de falha foram escalonados em número maior que o da conectividade da rede.

Os experimentos foram executados em quatro tipos de topologias. O conjunto de topologias é mostrado na Tabela 2. Primeiramente, grafos com conectividade de vértices igual a k gerados aleatoriamente foram obtidos como em [Subbiah and Blough 2004]. Uma topologia inicial de n nodos com grau k foi obtida aleatoriamente. A conectividade de vértices da topologia foi então repetidamente avaliada. A cada vez em que a conectividade desejada não era atingida, n enlaces adicionais eram aleatoriamente introduzidos, até que uma topologia com conectividade de vértices igual a k era obtida. Topologias foram geradas para n igual a 32, 64, 128 e 256 nodos com $k = 3$ e $k = \log n$. Cinco diferentes topologias foram geradas para cada valor de n e k.

Topologias com distribuição *Power-Law* refletem a topologia da própria Internet, onde poucos nodos possuem muitos enlaces e a imensa maioria dos nodos possuem poucos enlaces. Grafos aleatórios com a distribuição *Power-Law* foram obtidas utilizando o gerador de [Bu and Towsley 2002]. Naquele gerador, o número total n de nodos e o número inicial de nodos no *backbone* são dados. A cada iteração, novos nodos são criados com dada probabilidade p. Com probabilidade $1-p$, novos enlaces são criados.

Tabela 2. Topologias utilizadas nas simulações

Topologias	
Grafos aleatórios com conectividade de vértices $k = 3$	32, 64, 128, 256 nodos
Grafos aleatórios com conectividade de vértices $k = \log n$	32, 64, 128, 256 nodos
Grafos aleatórios com distribuição *Power-Law*	32, 64, 128, 256 nodos
Topologias *Mesh*	64, 256 nodos
Hipercubos	32, 64, 128, 256 nodos

Neste caso, os nodos que serão conectados pelo novo enlace são escolhidos de acordo com um parâmetro, $\beta \in (-\infty, 1)$, o qual define o grau de preferência por nodos com muitos vizinhos. Quanto menor o valor de β, menor a preferência dada por um enlace a nodos com grau alto. Com $p = 0.6$ e $\beta = 0.2$, cinco diferentes topologias foram geradas para cada número n de 32, 64, 128 e 256 nodos. Estas topologias foram todas avaliadas como tendo conectividades de vértices e arestas iguais a 1, sendo, portanto, adequadas a experimentos com particionamento da rede.

Além destas, as topologias regulares *mesh 8x8* e *16x16* [Culler and Singh 1999] e hipercubos com 32, 64, 128 e 256 nodos foram também empregadas nos experimentos de simulação.

Como cinco diferentes topologias aleatórias de cada tipo foram geradas para cada número n de nodos, uma rodada foi simulada em cada topologia, e as médias das cinco rodadas foram testadas para a precisão do intervalo de confiança desejada. Com topologias regulares, como apenas uma topologia de cada tipo existe para cada número n de nodos, cinco rodadas independentes foram obtidas variando a semente para geração de números aleatórios em cada rodada.

Os experimentos são apresentados a seguir em duas seções; na segunda seção são apresentados resultados de simulação com particionamentos da rede.

4.1. Experimentos sem Particionamento da Rede

O primeiro conjunto de experimentos de simulação foi executado para propósitos de comparação com o algoritmo *ForwardHeartbeat*. Primeiramente, eventos de falha e de recuperação ocorrendo exclusivamente em nodos foram simulados com ambas as médias da distribuição exponencial de 1 segundo e de 200 segundos, em topologias com conectividade de vértices igual a 3. Os resultados da latência de diagnóstico são mostrados nas Figuras 1 e 2. Após isso, eventos de falha e de recuperação ocorrendo em nodos foram simulados em experimentos com média de 200 segundos para o processo de Poisson em ambas as redes com conectividade de vértices iguais a 3 e $\log n$. Os resultados são mostrados nas Figuras 3 e 4.

Como pode ser observado, as latências médias não são influenciadas nem pela conectividade da rede nem pelo dinamismo da ocorrência de eventos dado pelo processo de Poisson. Latências médias para eventos de falha são da ordem de um terço do intervalo de testes com conectividades da rede e médias da distribuição exponencial variadas. As latências médias do *ForwardHeartbeat* são da ordem de dois terços do intervalo entre *heartbeats* nas mesmas condições. As latências máximas do DNR, por outro lado, mostram um acréscimo em redes pequenas. Isto é devido ao fato de que, com poucos nodos, é maior a probabilidade de que eventos de falha em nodos contíguos ocorram. Assim, é

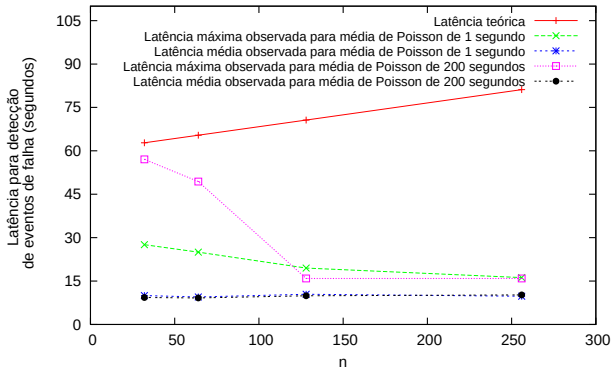


Figura 1. Eventos de falha em nodos - conectividade de vértices igual a 3 - grafos aleatórios.

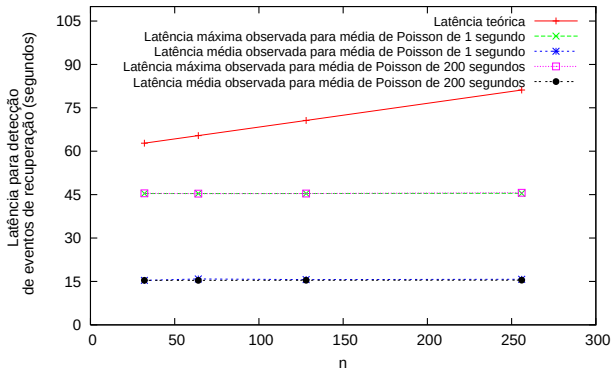


Figura 2. Eventos de recuperação em nodos - conectividade de vértices igual a 3 - grafos aleatórios.

possível que o próximo testador em um intervalo de testes esteja falho quando seu vizinho também falha. Com uma média de 200 segundos para o processo de Poisson, aquele testador permanece falho por mais tempo. Então pode levar até o máximo de dois intervalos de teste para que a falha do nodo vizinho, descrita acima, seja detectada por outro nodo.

Em experimentos de recuperação, a latência máxima ocorre em casos de testes simultâneos. Como estes casos são raros, eles não influenciam as latências médias, que são mantidas na ordem da metade do intervalo de testes. Além disto, em oposição aos experimentos com *ForwardHeartbeat*, a latência é calculada levando em conta o *recovery wait time*. De fato, tão logo expira o tempo de $W = 15.026516$ segundos, cada recuperação de um nodo é detectada. Se o *recovery wait time* não fosse levado em conta, as latências de recuperação do DNR também seriam da ordem de décimos de segundos, i.e. da ordem apenas do tempo de disseminação de cada evento. As latências médias também não crescem com o acréscimo no número de nodos na rede porque o diâmetro da rede mantém-se pequeno em comparação com n. O diâmetro máximo observado é de 25 em uma rede de 256 nodos e é da ordem de 7 em redes com 32 nodos.

Em ambos os tipos de experimentos, a latência teórica não muda com diferentes parâmetros de simulação ou tipos de eventos e é uma função do número n de nodos na rede.

Um segundo conjunto de experimentos com eventos em nodos foi executado em

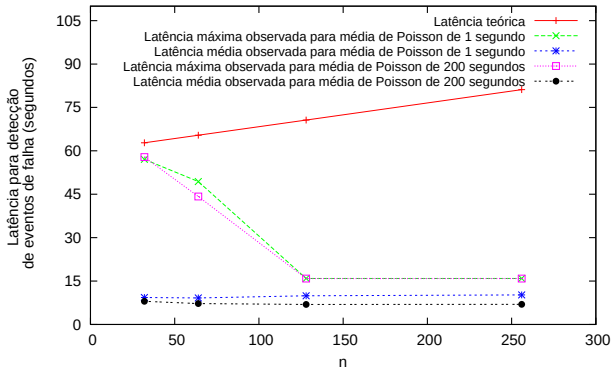


Figura 3. Eventos de falha em nodos - média de Poisson de 200 segundos - grafos aleatórios.

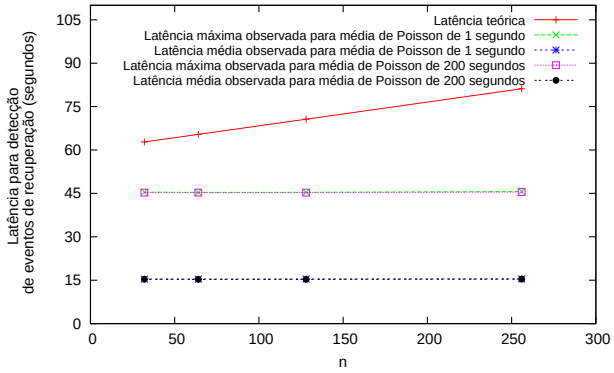


Figura 4. Eventos de recuperação em nodos - média de Poisson de 200 segundos - grafos aleatórios.

topologias regulares. As topologias foram redes *mesh* e hipercubos, e novamente as médias para o processo de Poisson foram de 1 segundo e 200 segundos. A conectividade de vértices de redes *mesh* é 4 e a conectividade de vértices de hipercubos é $\log n$. Um número máximo de nodos igual à conectividade menos 1 esteve falho simultaneamente em dado instante de tempo.

As latências médias de eventos de falha são novamente da ordem de um terço do intervalo de testes, como pode ser visto nas Figuras 5 e 6. Novamente as latências máximas mostram um acréscimo em redes pequenas para experimentos com média da distribuição exponencial de 200 segundos. Com média da distribuição exponencial igual a 1 segundo, a latência máxima apresenta um pequeno acréscimo com o acréscimo do tamanho da rede.

Os gráficos dos experimentos de *healing* exibem exatamente as mesmas curvas que aqueles do primeiro conjunto de experimentos acima e portanto não serão mostrados aqui. De fato, como um evento de recuperação de um nodo é detectado tão logo aquele nodo conclua o *recovery wait time*, nenhuma mudança na topologia influencia estes resultados, desde que o nodo em recuperação tenha ao menos um vizinho para detectar o evento.

Um outro conjunto de experimentos de simulação foi executado nas mesmas to-

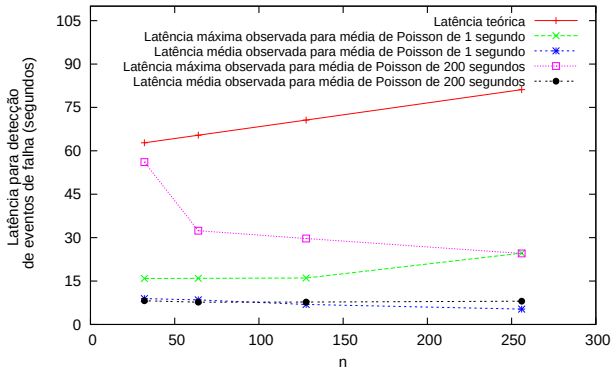


Figura 5. Eventos de falha em nodos - hipercubos.

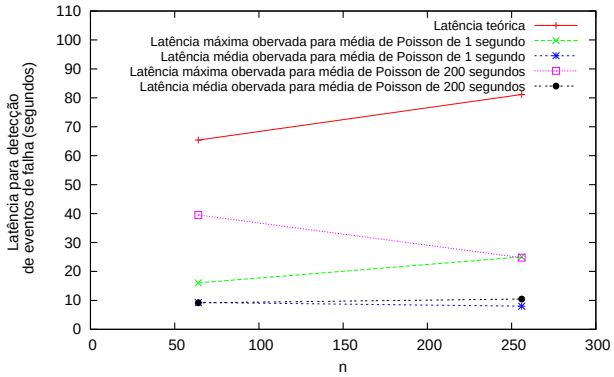


Figura 6. Eventos de falha em nodos - redes *mesh*.

pologias acima. Para este tipo de experimentos, eventos foram escalonados para ocorrer exclusivamente em enlaces. Os resultados espelham o que é teoricamente esperado: sem eventos ocorrendo em nodos, um enlace falho ou recentemente recuperado tem sempre um testador, e qualquer detecção ocorre em no máximo um intervalo de testes, independente do tipo de evento. As médias, tanto para eventos de falha como para eventos de recuperação, são iguais à metade do intervalo de testes. Esta regularidade é mantida tanto para redes aleatórias com ambas as conectividades de 3 e $\log n$ como para as redes *mesh* e os hipercubos. Os gráficos para topologias aleatórias são mostrados nas Figuras 7 e 8.

4.2. Experimentos com Particionamento da Rede

Experimentos com particionamento da rede foram executados em grafos aleatórios gerados com a distribuição *Power-Law*. Um componente conexo de uma rede pode sofrer particionamento quando um enlace ponte sofre um evento de falha ou quando um nodo falha. No primeiro caso, a rede fica particionada em dois componentes conexos, contendo cada um deles um dos nodos conectados pelo enlace recentemente falho. Um dos nodos, o próximo testador, detectará o evento em no máximo um intervalo de testes, enquanto o nodo que seria testado caso o enlace se mantivesse sem-falha detectará o evento no intervalo de testes seguinte. Por outro lado, a falha de um nodo, sobretudo daqueles com grande número de enlaces, pode criar vários componentes conexos. Em cada componente conexo, os nodos vizinhos ao nodo falho também detectarão eventos de falha em seus enlaces em um ou dois intervalos de teste, conforme a configuração da rede no momento

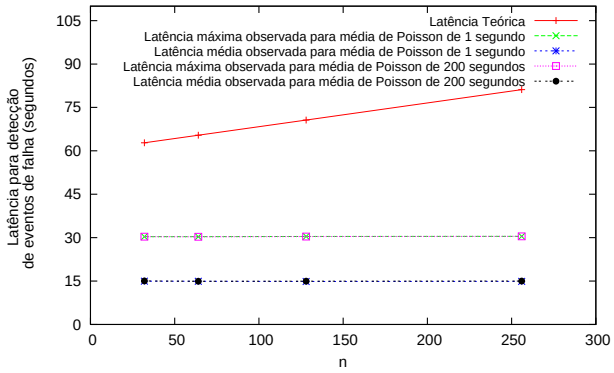


Figura 7. Eventos de falha em enlaces - conectividade de vértices igual a $\log n$ - grafos aleatórios.

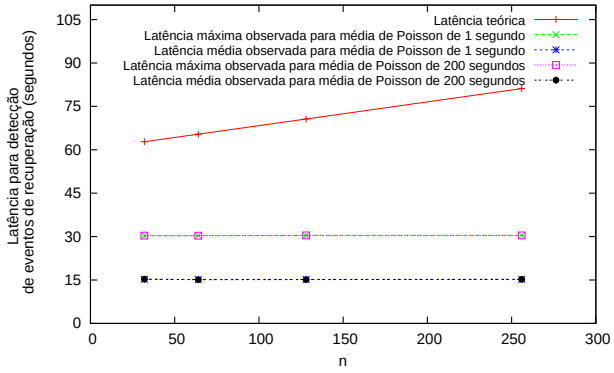


Figura 8. Eventos de recuperação em enlaces - conectividade de vértices igual a $\log n$ - grafos aleatórios.

da falha. Como a falha de um nodo pode ser simulada pela falha em todos os seus enlaces, somente eventos em enlaces foram escalonados neste tipo de experimento. O número de eventos escalonado foi sempre maior que o da conectividade da rede, porém não tão grande que a rede se reduzisse a uma rede trivial.

Foram medidas nestes experimentos apenas as latências de diagnóstico de eventos de falha. Latências de eventos de recuperação não foram medidas, por estarem suficientemente representadas nos experimentos anteriores. Para cada evento, foram medidas a latência de diagnóstico ocorrida no primeiro intervalo de testes, em um dos novos componentes conexos, e a latência de diagnóstico ocorrida para o mesmo evento no segundo intervalo de testes, no outro componente conexo. Ambas as latências foram computadas separadamente para cada evento ocorrido, e as médias das latências do diagnóstico de eventos ocorridos em cada intervalo de testes são mostradas na Figura 9. A média utilizada no processo de Poisson foi de 200 segundos.

Como num grafo aleatório gerado com a distribuição *Power-Law* um grande número de enlaces ponte são enlaces de nodos de borda, em muitos casos a disseminação de mensagens de diagnóstico fica confinada a componentes conexos com apenas um nodo. Isto é compensado pelas disseminações que ocorrem nos componentes conexos formados pelo restante da rede e, na média, as latências exibem o mesmo padrão de comportamento encontrado nos experimentos com falhas em enlace. Desta forma, tanto as

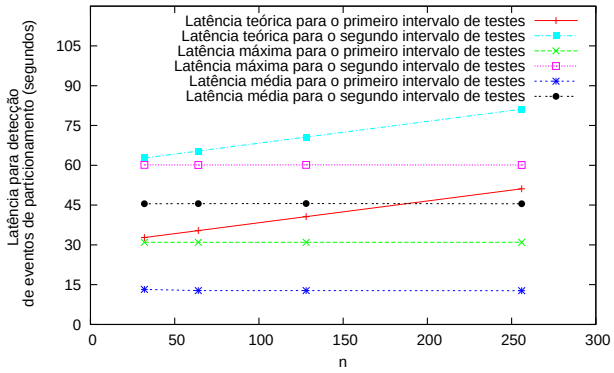


Figura 9. Eventos de falha em enlaces que particionam a rede - grafos *powerlaw*.

latências médias de diagnóstico ocorrido no primeiro como no segundo intervalo de testes se mantém na metado do intervalo de testes correspondente.

5. Conclusão

Este trabalho apresentou a avaliação do algoritmo *Distributed Network Reachability* através de técnicas de simulação de eventos discretos. Os experimentos foram realizados no *cluster* de alto desempenho da Universidade Federal do Paraná. O desempenho do algoritmo foi avaliado em várias topologias aleatórias e regulares, levando em conta múltiplos eventos de falha e recuperação, tanto de nodos como de enlaces. Situações na qual a rede sofre particionamento e na qual ela permanece sempre conectada foram avaliadas. Um conjunto de experimentos foi conduzido de forma a comparar o desempenho dos algoritmos DNR e *ForwardHeartbeat*. O dinamismo da ocorrência de eventos foi dado por processos de Poisson, com médias de 1 segundo e 200 segundos. Para obter um intervalo de confiança de 95% para as médias das latências de diagnóstico, para cada tipo e tamanho de topologia 5000 eventos de falha e recuperação (2500 de cada tipo) foram escalonados e executados em 5 rodadas independentes.

Referências

Bagchi, A. and Hakimi, S. L. (1991). "An Optimal Algorithm for Distributed System-Level Diagnosis," *Proc.* 21^{st} *Int'l Symp. Fault Tolerant Computing*, pp. 214-221, 1991.

Bu, T. and Towsley, D. (2002) "On Distinguishing between Internet Power Law Topology Generators," *IEEE INFOCOM*, vol. 2, pp 638-647, 2002.

Chessa, S. and Santi, P. (2001) "Comparison-Based System-Level Fault Diagnosis in Ad Hoc Networks," *Proc. 20th IEEE Symp. on Reliable Distributed Systems*, 2001.

Chessa, S. and Santi, P. (2002) "Crash Faults Identification in Wireless Sensor Networks," *Computer Communications*, Vol. 25, No. 14, Sept. 2002.

Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C. (2001) *Introduction to Algorithms*, The MIT Press, 2001.

Culler, D. E. and Singh, J. P. (1999) *Parallel Computer Architecture - A Hardware/Software Approach*, Morgan Kaufmann, 1999.

Ding, M., Chen, D., Xing, K. and Cheng, X. (2005) "Localized Fault-Tolerant Event Boundary Detection in Sensor Networks," *Proc. 24th Annual IEEE Conf. Computer and Communication Societies*, 2005.

Duarte Jr., E. P., Mansfield, G., Nanya, T. and Noguchi, S. (1997) "Non-Broadcast Network Fault Monitoring Based on System-Level Diagnosis," *Proc. IEEE/IFIP IM'97*, pp.597-609, San Diego, May 1997.

Duarte Jr., E. P. and Weber, A. (2003) "A Distributed Network Connectivity Algorithm," *Proc. IEEE/ISADS'03*, pp.285-292, Pisa, April 2003.

Elhadef, M., Boukerche, A. and Elkadiki, H. (2004) "An Adaptive Fault Identification Protocol for an Emergency/Rescue-Based Wireless and Mobile Ad Hoc Network," *Proc. Symp. Parallel and Distributed Processing*, pp.1-8, 2007.

Khanna, G., Cheng, M. Y., Varadharajan, P., Bagchi, S., Correia, M. P. and Veríssimo, P. J. (2007) "Automated Rule-Based Diagnosis through a Distributed Monitor System," *IEEE Transactions on Dependable and Secure Computing*, Vol. 4, No. 4, pp. 266-279, October-December 2007.

Lee, M. H. and Choi, Y. H. (2007) "Distributed Diagnosis of Wireless Sensor Networks," *Proc. IEEE Region 10 Conference*, pp.1-4, 2007.

MacDougall, M. H. (1987) *Simulating Computer Systems: Techniques and Tools*, The MIT Press, Cambridge, MA, 1987.

Rangarajan, S., Dahbura, A. T. and Ziegler, E. A. (1995) "A Distributed System-Level Diagnosis Algorithm for Arbitrary Network Topologies," *IEEE Trans. Computers*, Vol.44, pp. 312-333, 1995.

Santi, P. and Blough, D. M. (2002) "An Evaluation of Connectivity in Mobile Wireless Ad Hoc Networks," *Proc. IEEE Int'l. Conf. Dependable Systems and Networks*, pp. 89-102, Washington, 2002.

Stahl, M., Buskens, R. and Bianchini, R. (1992) "Simulation of the Adapt On-Line Diagnosis Algorithm for General Topology Networks," *Proc. IEEE* 11^{th} *Symp. Reliable Distributed Systems*, October 1992.

Subbiah, A. and Blough, D. M. (2004) "Distributed Diagnosis in Dynamic Fault Environments," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 15, No. 5, pp. 453-467, May 2004.

Weber, A. (2008) *Um Algoritmo de Diagnóstico Distribuído para Redes Particionáveis de Topologia Arbitrária* Tese de Doutorado, UTFPR, 2008. (previsão de defesa - maio 2008)

Weber, A., Duarte Jr., E. P. and Fonseca, K. O. (2006) "An Optimal Test Assignment for Monitoring General Topology Networks," *Proc. 7th IEEE Latin-American Test Workshop*, pp. 131-136, Buenos Aires, 2006