XXX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos

30 de abril a 4 de maio de 2012
Ouro Preto, MG

XIII Workshop de Testes e Tolerância a Falhas (WTF)

Editora
Sociedade Brasileira de Computação (SBC)

Organizadores
Fabíola Gonçalves Pereira Greve (UFBA)
Aldri Luiz dos Santos (UFPR)
Jussara Marques de Almeida (UFMG)
Dorgival Olavo Guedes Neto (UFMG)

Realização
Departamento de Ciência da Computadação
Universidade Federal de Minas Gerais (UFMG)

Promoção
Sociedade Brasileira de Computação (SBC)
Laboratório Nacional de Redes de Computadores (LARC)

Copyright © 2012 da Sociedade Brasileira de Computação
Todos os direitos reservados

Capa: Centro de Comunicações (Cedecom) da UFMG
Produção Editorial: Jussara Marques de Almeida e Dorgival Olavo Guedes Neto

Cópias Adicionais:
Sociedade Brasileira de Computação (SBC)
Av. Bento Gonçalves, 9500 – Setor 4 – Prédio 43.412 - Sala 219
Bairro Agronomia – CEP 91.509-900 – Porto Alegre – RS
Fone: (51) 3308-6835
E-mail: sbc@sbc.org.br

Workshop de Testes e Tolerância a Falhas (13 : 2012: Ouro Preto, MG)

W926a Anais / XIII Workshop de Testes e Tolerância a Falhas ; organizado por Jussara Marques de Almeida...[et al.] — Porto Alegre: SBC, 2012.
146 p. : il.

Conhecido também: WTF 2012.
Realização: Universidade Federal de Minas Gerais.

ISSN: 2177-496X

1. Redes de Computadores - Congressos. 2. Sistemas Distribuídos - Congressos. I. Almeida, Jussara Marques de. II. Sociedade Brasileira de Computação. III.Título

CDU 004.7(063)

Promoção

Sociedade Brasileira de Computação (SBC)

Diretoria

Presidente
Paulo Roberto Freire Cunha (UFPE)

Vice-Presidente
Lisandro Zambenedetti Granville (UFRGS)

Diretor Administrativo
Luciano Paschoal Gaspary (UFRGS)

Diretora de Finanças
Luci Pirmez (UFRJ)

Diretor de Eventos e Comissões Especiais
Altigran Soares da Silva (UFAM)

Diretora de Educação
Mirella Moura Moro (UFMG)

Diretora de Publicações
Karin Koogan Breitman (PUC-Rio)

Diretora de Planejamento e Programas Especiais
Ana Carolina Brandão Salgado (UFPE)

Diretora de Secretarias Regionais
Thais Vasconcelos Batista (UFRN)

Diretor de Divulgação e Marketing
Edson Norberto Cáceres (UFMS)

Diretor de Relações Profissionais
Roberto da Silva Bigonha (UFMG)

Diretor de Competições Científicas
Ricardo de Oliveira Anido (UNICAMP)

Diretor de Cooperação com Sociedades Científicas
Raimundo José de Araújo Macêdo (UFBA)

Promoção

Conselho

Mandato 2011-2015
Ariadne Carvalho (UNICAMP)
Carlos Eduardo Ferreira (IME - USP)
José Carlos Maldonado (ICMC - USP)
Luiz Fernando Gomes Soares (PUC-Rio)
Marcelo Walter (UFRGS)

Mandato 2009-2013
Virgílio Almeida (UFMG)
Flávio Rech Wagner (UFRGS)
Silvio Romero de Lemos Meira (UFPE)
Itana Maria de Souza Gimenes (UEM)
Jacques Wainer (UNICAMP)

Mandato 2011-2013
César A. F. De Rose (PUCRS)
Maria Izabel Cavalcanti Cabral (UFCG)
Renata Mendes de Araújo (UNIRIO)
Ricardo Augusto da Luz Reis (UFRGS)

Laboratório Nacional de Redes de Computadores (LARC)

Diretoria

Diretor do Conselho Técnico-Científico
Luciano Paschoal Gaspary (UFRGS)

Diretor Executivo
Célio Vinicius Neves de Albuquerque (UFF)

Vice-Diretor do Conselho Técnico-Científico
Artur Ziviani (LNCC)

Vice-Diretor Executivo
Elias P. Duarte Jr. (UFPR)

Membros Institucionais
CEFET-CE, IME, INPE/MCT, LNCC, PUCPR, PUC-Rio, PUC-RS, SESU/MEC, UECE, UFABC, UFAM, UFBA, UFC, UFCG, UFES, UFF, UFMG, UFMS, UFPA, UFPE, UFPR, UFRGS, UFRJ, UFRN, UFS, UFSC, UFSCar, UNB, UNICAMP, UNIFACS, UNIRIO, USP e UTFPR.

Realização

Comitê de Organização

Coordenadores Gerais

Jussara Marques de Almeida (UFMG)

Dorgival Olavo Guedes Neto (UFMG)

Coordenação do Comitê de Programa

Elias P. Duarte Jr. (UFPR)

José F. de Rezende (UFRJ)

Coordenação de Palestras e Tutoriais

Edmundo Madeira (UNICAMP)

Coordenação de Painéis e Debates

Francisco V. Brasileiro (UFCG)

Coordenação de Minicursos

Antônio Jorge Gomes Abelém (UFPA)

Coordenação de Workshops

Aldri Luiz dos Santos (UFPR)

Coordenadores do Salão de Ferramentas

Ricardo Augusto Rabelo Oliveira (UFOP)

Humberto Torres Marques Neto (PUC Minas)

Comitê Consultivo

Antônio Jorge Gomes Abelém (UFPA)

Artur Ziviani (LNCC)

Bruno Schulze (LNCC)

Célio Albuquerque (UFF)

Luci Pirmez (UFRJ)

Luciano Paschoal Gaspary (UFRGS)

Marinho Pilla Barcellos (UFRGS)

Ronaldo Alves Ferreira (UFMS)

Thais Vasconcelos Batista (UFRN)

Realização

Organização Local

Antonio Alfredo F. Loureiro (UFMG)

Carlos Frederico Marcelo da Cunha Cavalcanti (UFOP)

Cristina Duarte Murta (CEFET- MG)

Daniel Fernandes Macedo (UFMG)

Dorgival Guedes Neto (UFMG)

Fabrício Benevenuto (UFOP)

Flávia Otoni (UFMG)

Humberto Torres Marques Neto (PUC Minas)

Ítalo Cunha (UFMG)

José Marcos Nogueira (UFMG)

Jussara M. Almeida (UFMG)

Luis Filipe M. Vieira (UFMG)

Marcos Vieira (UFMG)

Murilo Silva Monteiro (UFMG)

Olga Goussevskaia (UFMG)

Raquel Mini (PUC Minas)

Ricardo A. Rabelo (UFOP)

Mensagem dos Coordenadores Gerais

Sejam bem vindos ao XXX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC 2012) na bela e histórica cidade de Ouro Preto, MG! Promovido anualmente pela Sociedade Brasileira de Computação (SBC) e pelo Laboratório Nacional de Redes de Computadores (LARC), o SBRC 2012, em sua trigésima edição, busca manter a sua posição como o mais importante evento científico nacional em Redes de Computadores e Sistemas Distribuídos e um dos maiores da área de Informática no país. Ele busca também, como é sua tradição, estimular a troca de ideias e experiências, a discussão de temas de pesquisa desafiadores, assim como a aproximação entre estudantes, professores, pesquisadores e profissionais.

Para tal, o SBRC 2012 está com uma programação bastante rica, variada e com alta qualidade técnica. São 19 sessões técnicas com a apresentação de artigos completos cobrindo uma ampla lista de tópicos bastante relevantes e atuais nas áreas de Redes de Computadores e Sistemas Distribuídos; 3 sessões técnicas com a apresentação de ferramentas; 5 minicursos, com quatro horas de duração cada, sobre temas atuais normalmente não abordados nas grades curriculares; 5 palestras e 1 tutorial proferidos por pesquisadores de alto renome internacional, 3 palestras de profissionais de empresas de reconhecida excelência e liderança no mercado e 3 painéis versando sobre temas atuais e polêmicos. Além dessas atividades, 8 workshops ocorrerão em paralelo com o evento: XVII Workshop de Gerência e Operação de Redes e Serviços (WGRS), XIII Workshop da Rede Nacional de Ensino e Pesquisa (WRNP), XIII Workshop de Testes e Tolerância a Falhas (WTF), X Workshop de Computação em Grade e Aplicações (WCGA), VIII Workshop de Redes Dinâmicas e Sistemas P2P (WP2P), III Workshop de Pesquisa Experimental na Internet do Futuro (WPEIF), II Workshop on Autonomic Distributed Systems (WoSiDA) e II Workshop de Redes de Acesso em Banda Larga (WRA).

Em particular, em comemoração às suas 30 edições, o SBRC 2012 traz algumas novidades. A programação técnica conta com uma sessão de artigos convidados, escritos por expoentes da comunidade. Estes artigos vêm relembrar a história do evento e da sua comunidade científica, fomentando uma reflexão sobre a sua evolução ao longo dessas 30 edições e sobre novos rumos para essa comunidade. A programação também conta com um painel que visa fazer uma retrospectiva crítica dos 30 anos de pesquisa em Redes no Brasil, contrastando-a com a evolução da área globalmente. Por fim, para homenagear aqueles que contribuíram significativamente para o desenvolvimento da pesquisa e o fortalecimento da comunidade científica nas áreas de Redes de Computadores e Sistemas Distribuídos no Brasil, o SBRC 2012 cria o Prêmio Destaque SBRC, que passará a ser concedido anualmente. Nesta primeira edição do prêmio, o homenageado é o Professor Edmundo Albuquerque de Souza e Silva, da Universidade Federal do Rio de Janeiro, em reconhecimento às suas contribuições científicas e aos serviços prestados em benefício do SBRC e de sua comunidade.

A organização de um evento do porte e da importância do SBRC só pode ser realizada se contar com a ajuda de uma equipe capacitada e coesa. Gostaríamos de agradecer aos membros dos Comitês de Organização Geral e Local pelo trabalho voluntário de excelente qualidade e pelo apoio incansável durante as várias etapas da organização

Mensagem dos Coordenadores Gerais

deste evento. Somos muito gratos também ao apoio prestado pela SBC, particularmente aos membros da Comissão Especial de Redes de Computadores e Sistemas Distribuídos, pela prontidão e atenção que nos foram dedicadas. Gostaríamos de agradecer ainda ao Comitê Gestor da Internet no Brasil (CGI.br), às agências governamentais de fomento – CNPq, CAPES e FAPEMIG, ao Instituto Nacional de Ciência e Tecnologia para a Web (INWeb) e aos patrocinadores por reconhecerem e valorizarem a importância do SBRC como fórum de divulgação de pesquisa e inovação. Por fim, nossos agradecimentos ao Departamento de Ciência da Computação da UFMG, por viabilizar a realização deste evento, assim como ao Departmento de Computação da UFOP, pelo apoio a essa realização.

Em nome do comitê organizador do SBRC 2012, desejamos a todos uma semana bastante produtiva e agradável. Esperamos que a tradicional hospitalidade mineira e que a fabulosa e setecentista Ouro Preto deixem lembranças inesquecíveis desta trigésima edição do SBRC.

Jussara Marques de Almeida
Dorgival Olavo Guedes Neto
Coordenadores Gerais do SBRC 2012

Mensagem do Coordenador de Workshops do SBRC 2012

Os workshops são uma parte tradicional do que hoje faz do SBRC o principal evento da área no país, sendo responsáveis por atrair uma parcela cada vez mais expressiva de participantes para o Simpósio. Este ano damos continuidade a iniciativa de chamada aberta de workshops proposta no SBRC 2011. A chamada de workshops estimula a participação da comunidade de Redes e Sistemas Distribuídos a sugerir e fortalecer novas linhas de pesquisa, bem como manter em evidência linhas de pesquisas tradicionais.

Em resposta à chamada, recebemos nove propostas de alta qualidade, das quais sete foram aceitas e seus respectivos proponentes convidados a organizarem os workshops no SBRC em Ouro Preto. Das propostas aceitas, quatro delas tratavam de workshops já tradicionais no SBRC e considerados parte do circuito nacional de divulgação científica nas várias subáreas de Redes de Computadores e Sistemas Distribuídos, como o WGRS (Workshop de Gerência e Operação de Redes e Serviços), o WTF (Workshop de Testes e Tolerância a Falhas), o WCGA (Workshop em Clouds, Grids e Aplicações) e o WP2P (Workshop de Redes Dinâmicas e Sistemas P2P). Uma proposta referia-se à terceira edição do WPEIF (Workshop de Pesquisa Experimental da Internet do Futuro). E as outras duas propostas, a segunda edição do WRA (Workshop de Redes de Acesso em Banda Larga) e do WoSiDA (Workshop de Sistemas Distribuídos Autonômicos), consolidando assim novas linhas de pesquisa dentro da nossa comunidade científica.

Esperamos que 2012 seja mais um ano de sucesso para os workshops do SBRC, contribuindo como importantes fatores de agregação para os avanços promovidos pela comunidade científica da área de Redes e Sistemas Distribuídos no Brasil.

Aproveitamos para agradecer o inestimável apoio recebido de diversos membros da comunidade e, em particular a cada coordenador de workshop pelo brilhante trabalho e a Organização Geral do SBRC 2012.

A todos, um excelente SBRC em Ouro Preto!

Aldri Luiz dos Santos
Coordenador de Workshops do SBRC 2012

Mensagem da Coordenadora do WTF 2012

Bem vindos ao XIII Workshop de Testes e Tolerância a Falhas (WTF). O WTF constitui-se num fórum de debates e apresentações de trabalhos teóricos e práticos, desenvolvidos principalmente pela comunidade brasileira, em pesquisas relacionadas à confiança no funcionamento de sistemas, em especial nas áreas de testes e tolerância a falhas. O espaço promovido pelo WTF é importante para divulgar resultados recentes e identificar perspectivas de pesquisa futuras, bem como promover a integração entre os diversos membros da academia e indústria brasileiras.

O WTF surgiu em 1998 como um evento suplementar ao Simpósio Brasileiro de Computação Tolerante a Falhas (SCTF). A partir de 2003, o SCTF foi substituído por um evento de caráter internacional, o *Latin American Symposium on Dependable Computing* (LADC). Desde então, o WTF costuma ser organizado como um evento satélite, ora do LADC, ora do SBRC - *Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*.

Nesta edição de 2012, o WTF ocorrerá em conjunto com o SBRC e contemplará atividades de apresentação de artigos, além de palestra convidada. Foram submetidos 21 trabalhos ao evento, sendo 16 artigos regulares, 3 relatórios de experiências práticas e 2 resumos estendidos. Destes, foram selecionados 9 artigos regulares e 1 relatório prático. Cada um deles teve no mínimo 3 revisões, feitas por um Comitê de Avaliação criterioso composto por 36 pesquisadores especialistas nas diversas áreas.

Os artigos, escolhidos pela sua qualidade e organizados em quatro sessões técnicas, denunciam temas de pesquisa atuais e concentram-se nas áreas de (1) injeção de falhas e avaliação de ataques, (2) protocolos distribuídos e avaliação de modelos, (3) testes, verificação e especificação formal, (4) além de cobertura de falhas. Nessa edição, o WTF irá outorgar um prêmio ao melhor artigo, dentre os melhor avaliados. A escolha foi feita por um conjunto de pesquisadores do Comitê de Avaliação sem conflitos de interesse com os autores e a sua nomeação ocorrerá durante a realização do Workshop.

Para enriquecer e fomentar mais ainda as discussões durante o evento, convidamos o professor Alysson Bessani, pesquisador do laboratório de pesquisa LASIGE, vinculado à Universidade de Lisboa. Ele nos dará uma palestra sobre aspectos de confiabilidade e segurança no uso da computação em nuvem, com especial atenção ao papel da replicação de dados e serviços usando múltiplos provedores de serviço na nuvem. Esperamos que os trabalhos e palestra apresentados suscitem discussões e interações frutuosas e contribuam mais ainda para o desenvolvimento da área no Brasil.

Aproveito para agradecer a todo o Comitê de Programa e Revisores pelo excelente trabalho de avaliação realizado, pela reatividade e boas discussões, que contribuíram para termos uma seleção criteriosa e construtiva. Agradecemos aos autores e ao palestrante convidado pelas suas valiosas contribuições. Agradecemos a Comissão Organizadora do SBRC 2012 por nos acolher e fornecer o apoio logístico para a concretização do evento. Finalmente, um agradecimento especial ao Aldri Santos, Coordenador de Workshops, pela presteza e apoio constantes ao nosso trabalho.

Fabíola Gonçalves Pereira Greve
Coordenadora do WTF 2012

Comitê de Programa do WTF

Alcides Calsavara (PUCPR)
Alexandre Sztajnberg (UERJ)
Alfredo Goldmann (USP)
Alysson Bessani (Univ. de Lisboa)
Ana Maria Ambrosio (INPE)
Avelino Zorzo (PUCRS)
Carlos Montez (UFSC)
Daniel Batista (USP)
Daniel Macêdo (UFMG)
Daniel Mossé (Univ. of Pittsburgh)
Eduardo Bezerra (UFSC)
Eliane Martins (UNICAMP)
Elias P. Duarte Jr. (UFPR)
Fabíola Greve (UFBA)
Fernando Dotti (PUCRS)
Fernando Pedone (Univ. of Lugano)
Francisco Brasileiro (UFCG)
Frank Siqueira (UFSC)
Irineu Sotoma (UFMS)
José Pereira (Univ. do Minho)
Lásaro Camargos (UFU)
Lau Cheuk Lung (UFSC)
Lúcia Drummond (UFF)
Luciana Arantes (Univ. de Paris VI)
Marco Vieira (Univ. de Coimbra)
Marinho Barcellos (UFRGS)
Michele Nogueira (UFPR)
Miguel Correia (Univ. de Lisboa)
Patrícia Machado (UFCG)
Raimundo Barreto (UFAM)
Raul Ceretta Nunes (UFSM)
Regina Moraes (UNICAMP)
Rogério de Lemos (Univ. de Kent)
Rui Oliveira (Univ. do Minho)
Taisy Weber (UFRGS)
Udo Fritzke Jr. (PUC Minas)

Revisores do WTF

Alcides Calsavara (PUCPR)
Alexandre Sztajnberg (UERJ)
Alfredo Goldmann (USP)
Alysson Bessani (Univ. de Lisboa)
Ana Maria Ambrosio (INPE)
Avelino Zorzo (PUCRS)
Carlos Montez (UFSC)
Daniel Batista (USP)
Daniel Macedo (UFMG)
Daniel Mossé (Univ. of Pittsburgh)
Djones Lettnin (UFSC)
Eduardo Bezerra (UFSC)
Eliane Martins (UNICAMP)
Elias P. Duarte Jr. (UFPR)
Fabíola Greve (UFBA)
Fernando Dotti (PUCRS)
Fernando Pedone (Univ. of Lugano)
Flávio Silva (UFU)
Francisco Brasileiro (UFCG)
Frank Siqueira (UFSC)
Irineu Sotoma (UFMS)
José Pereira (Univ. do Minho)
Kelly Braghetto (USP)
Lásaro Camargos (UFU)
Lau Cheuk Lung (UFSC)
Lúcia Drummond (UFF)
Luciana Arantes (Univ. de Paris VI)
Marco Vieira (Univ. de Coimbra)
Marinho Barcellos (UFRGS)
Michele Nogueira (UFPR)
Miguel Correia (Univ. de Lisboa)
Patrícia Machado (UFCG)
Raimundo Barreto (UFAM)
Raul Ceretta Nunes (UFSM)
Regina Moraes (UNICAMP)
Rogério de Lemos (Univ. of Kent)
Rui Oliveira (Univ. do Minho)
Taisy Weber (UFRGS)
Udo Fritzke Jr. (PUC Minas)

Sumário

Sessão Técnica 1 - Injeção de Falhas e Avaliação de Ataques 1

Aplicando Cadeias de Markov para Injeção de Perdas de Pacotes no Sistema Android
Alexandre Felin Gindri (UFRGS),Taisy Silva Weber (UFRGS) e Sérgio Luis Cechin (UFRGS) 3

Emulação de Ataques do Tipo XPath Injection para Testes de Web Services Usando Injeção de Falhas
Marcelo I. P. Salas (UNICAMP) e Eliane Martins (UNICAMP) 17

Análise do Impacto de Ataques de Poluição Combinado com Whitewashing em Sistemas P2P de Live Streaming
Rafael B. de Almeida (UFJF), Alex Borges Vieira (UFJF) e Ana Paula Couto Silva (UFJF) 31

Sessão Técnica 2 - Protocolos Distribuídos e Avaliação de Modelos 41

Exclusão Mútua Distribuída e Robusta para *k* Recursos Compartilhados
Luiz A. Rodrigues (UNIOESTE/UFPR), Elias P. Duarte Jr. (UFPR) e Luciana Arantes (Université de Paris VI) 43

Memória Compartilhada em Sistemas Bizantinos Dinâmicos
Eduardo A. P. Alchieri (UFSC), Alysson N. Bessani (Universidade de Lisboa), Joni da Silva Fraga (UFSC) e Fabíola Greve (UFBA) 57

Caracterização de uma Jornada Máxima em MANETs
Alex Novaes de Santana (UFBA), Fabíola Greve (UFBA) e José Garcia Vivas Miranda (UFBA) 71

Sumário

Sessão Técnica 3 - Testes, Verificação e Especificação Formal 85

Uma Abordagem para Geração de Casos de Teste Estrutural Baseada em Modelos
Leandro T. Costa (PUCRS), Flávio M. Oliveira (PUCRS), Elder M. Rodrigues (PUCRS), Maicon B. da Silveira (PUCRS) e Avelino F. Zorzo (PUCRS) ……...… 87

Modeling Communication Semantics for Distributed Systems in Event-B
Fernando Luís Dotti (PUCRS) e Leila Ribeiro (UFRGS) ……..... 101

Um Verificador de Modelos Descritos em Redes de Autômatos Estocásticos
Claiton M. Correa (PUCRS), Fernando L. Dotti (PUCRS), Paulo Fernandes (PUCRS), Eli Maruani (PUCRS), Lucas G. Oleksinski (PUCRS) e Afonso Sales (PUCRS) …….. 115

Sessão Técnica 4 - Cobertura de Falhas e Replicação na Cloud 129

Detecção de Falhas em Autômatos Grid
Gilson Doi Júnior (Universidade Estadual de Londrina) e Adilson Luiz Bonifácio (Universidade Estadual de Londrina) …… 131

Palestra Convidada: "Replicação na Cloud (of Clouds)"
Alysson N. Bessani (Universidade de Lisboa) …….. 145

Índice por Autor 146

XIII Workshop de Testes e Tolerância a Falhas

♦

Sessão Técnica 1

Injeção de Falhas e Avaliação de Ataques

Aplicando Cadeias de Markov para injeção de perdas de pacotes no sistema Android

Alexandre Felin Gindri[1], Taisy Silva Weber[1], Sérgio Luis Cechin[1]

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

{afgindri,taisy,cechin}@inf.ufrgs.br

Abstract. *We describe the development of a communication fault injector for mobile devices that allows the emulation of packet loss following fault models suitable for wireless networks. The fault injector named Mob-FI allows the test under faults of Java applications that execute on smartphones and tablets and that are based on UDP communication. We build the fault injector modifying the code of Dalvik Virtual Machine to emulate faults according to the models of packet loss represented as Markov Chains of up to four states.*

Resumo. *Descreve-se o desenvolvimento de um injetor de falhas de comunicação para dispositivos móveis que permite a emulação de perdas de pacotes seguindo modelos de falhas adequados a redes sem fio. O injetor Mob-FI permite o teste sob falhas de aplicações alvo Java desenvolvidas para smartphones e tablets e que sejam baseadas em comunicação UDP. O injetor foi construído modificando a Dalvik Virtual Machine do sistema Android e emula falhas de acordo com os modelos de perda de pacotes em rajada representados como Cadeias de Markov de até quatro estados.*

1 Introdução

O Android [Butler 2011], criado pela Google e pela Open Handset Alliance, facilita a criação de aplicações com acesso a todos os recursos de dispositivos móveis. Graças a essa facilidade, um grande número de programas, provenientes dos mais diversos desenvolvedores, está disponível para usuários de *smartphones* e *tablets*. Muitas dessas aplicações são usadas em situações em que se esperaria a garantia de um mínimo de confiabilidade e disponibilidade. Para fornecer tal garantia, seria desejável poder testar a tolerância dessas aplicações a uma variada gama de falhas que usualmente afeta a operação de dispositivos móveis em cenários de uso real. Entre as possíveis falhas, dispositivos móveis são especialmente sensíveis a falhas de comunicação, sendo comum pacotes serem perdidos durante a transmissão de dados. A perda de pacotes em ambientes móveis segue um comportamento típico desses ambientes, mas diferente daquela que ocorre em redes cabeadas.

Um desenvolvedor para Android que decida testar a robustez de uma aplicação em um cenário de falhas de comunicação deve determinar o modelo de falhas adequado ao ambiente, depois criar uma carga de falhas que corresponda a esse modelo e então executar a aplicação com uma dada carga de trabalho aplicando a carga de falhas. O objetivo deste trabalho é desenvolver um injetor de falhas que execute no ambiente Android, permita a injeção de perdas de pacotes seguindo modelos de falhas adequados ao ambiente móvel e facilite a descrição dos cenários de falhas para experimentos de teste a serem conduzidos por um desenvolvedor. Modelos de falhas para sistemas

móveis podem ser encontrados na literatura sendo que a perda de pacotes em rajada modelada como Cadeias de Markov com dois ou quatro estados [Salsano 2009] foram os escolhidos para descrição de carga de falhas.

As aplicações do Android rodam em uma máquina virtual própria do sistema, a Dalvik Virtual Machine [Bornstein 2009], que é uma máquina otimizada para sistemas com limitações de memória, processamento e bateria. Essa máquina é baseada na máquina virtual Java, mas se diferencia quanto ao formato dos bytecodes, não oferecendo apoio ao uso de mecanismos de instrumentação comumente usados para depuração e injeção de falhas.

Este artigo explora a possibilidade de implementar injetores de falhas de comunicação para o sistema Android a partir de modificação da máquina virtual. O injetor Mob-FI (Markov Fault Injetor for Mobile Environment) foi construído modificando a classe DatagramSocket da Dalvik Virtual Machine e emula falhas de acordo com os modelos de perda de pacotes representados como Cadeias de Markov. O injetor permite o teste sob falhas de aplicações móveis desenvolvidas em Java através da emulação de perda de pacotes de acordo com modelos de falhas de comunicação apropriados para redes sem fio.

Nas próximas seções, o artigo apresenta os conceitos básicos para a compreensão da injeção de falhas em ambientes móveis, discute modelos de falhas de comunicação em rajada, apresenta o desenvolvimento do injetor Mob-FI e experimentos conduzidos para avaliação do injetor.

2 Falhas de comunicação em ambientes móveis

A operação de dispositivos em ambientes móveis é caracterizada pela computação com recursos limitados, principalmente por tamanho e consumo de energia, e pela comunicação numa rede não cabeada, com movimentação relativa entre os dispositivos.

Com a popularização de equipamentos eletrônicos que operam nas mesmas faixas de frequências, aumenta também o problema da interferência entre sinais. A alteração na distância dos dispositivos devido à sua movimentação relativa causa mudanças na propagação dos sinais, acarretando variações, por exemplo, na atenuação, nos atrasos de recepção, na reflexão, refração e difração sofridas pelo sinal durante a propagação, entre outros.

Dispositivos móveis, portanto, estão sujeitos a diversas falhas de comunicação. A perda de pacotes pode ser causada por interferência ou movimentação. A falha pode ser transitória, quando ocorre perda de um único pacote. Podem ocorrer também rajadas de perda onde algumas mensagens em uma sequência de pacotes são perdidas. O dispositivo finalmente pode perder a conexão com a rede, sendo necessária uma renegociação com a sua estação base.

Como a comunicação é essencial em grande número de aplicações móveis, uma parte importante no teste destes sistemas é conseguir injetar falhas que emulem da forma mais apropriada possível esses cenários de perda de pacotes em rajada.

2.1 Construção de injetores de falhas de comunicação

Os injetores de falhas por software [Hsueh 1997] são injetores de erros, e podem alterar o comportamento de um programa injetando erros a partir de vários recursos de

software de um sistema. Várias falhas podem ser emuladas através de erros injetados, como, por exemplo, troca de bits em memória ou registradores, mas o foco deste artigo são as falhas que afetam a troca de mensagens através de uma rede de comunicação. Exemplos de injetores que permitem emular falhas de comunicação são Orchestra [Dawson 1996], NIST Net [Carson 2003], VirtualWire [De 2003], FIRMAMENT ([Drebes 2006][Siqueira 2009]), Fiona [Jacques-Silva 2006] e FAIL-FCI [Horau 2007]. Trabalhos anteriores do grupo finalizaram o porte do injetor FIRMAMENT para o sistema Android [Acker 2010]. Entre os injetores de falhas desenvolvidos no grupo de pesquisa, FIRMAMENT atua diretamente no kernel do sistema operacional e manipula todos os pacotes IP que trafegam pela pilha de protocolos no kernel, mesmo as mensagens que não estão relacionadas à aplicação alvo, o que em algumas situações pode invalidar o teste. Fiona e Comform [Menegotto 2011] atuam a partir da máquina virtual, e são ferramentas de injeção de falhas específicas para o teste sob falhas de aplicações alvo escritas em Java.

A forma usual de injetar falhas de comunicação é interceptar pacotes enviados e recebidos pela aplicação alvo e, uma vez de posse de uma mensagem, decidir, de acordo com uma dada descrição de carga de falhas, se a mensagem vai ser descartada, atrasada ou corrompida. Para interceptar pacotes, podem ser usados alguns recursos providos pelo sistema alvo como bibliotecas de comunicação, chamadas de sistema, reflexão computacional, módulos de depuração ou filtros disponíveis no kernel do sistema operacional. Os recursos de interceptação usados para o desenvolvimento dos injetores Comform e Fiona foram Javassist e JVMTI. Javassist permite a criação de classes durante a execução e a modificação de um arquivo de classe quando a máquina virtual for carregá-la. JVMTI é usada em ferramentas de desenvolvimento e monitoração. Permite inspecionar o estado e controlar a execução de aplicações em máquinas virtuais Java. Quando os estudos para a construção do injetor Mob-FI foram iniciados, nenhuma das duas opções estava disponível para a Dalvik Virtual Machine. Assim, uma alternativa viável foi modificar o código da máquina virtual.

2.2 Modificações no código fonte do sistema

A injeção de falhas por software consiste em inserir trechos de código que emulem a ocorrência de falhas e, após, retomar a operação com a falha injetada presente no sistema. As falhas podem ser inseridas diretamente na aplicação sob teste ou em alguma camada do sistema que executa a aplicação, como por exemplo, a máquina virtual, as bibliotecas do sistema e o próprio kernel do sistema operacional.

Editar diretamente o código do sistema para inserir trechos que emulem uma falha é uma alternativa em dispositivos com processadores que não tenham hardware de depuração apropriado para permitir o uso de ferramentas de inserção de código em execução [Looker 2005]. Uma vantagem do método é não ser necessário ter disponível o código da aplicação que será testada. A alteração do código do sistema apresenta, entretanto, desvantagens. O método pode não ser válido para uma eventual certificação, já que os sistemas com e sem o injetor podem ser considerados diferentes. Apesar disso, é possível desenvolver injetores com baixa interferência no desempenho e que podem ser aplicados a qualquer programa executado pelo sistema original.

3 Modelos de falhas de comunicação

O objetivo de injetar falhas em uma aplicação alvo é verificar se os mecanismos de tolerância a falhas serão suficientes e corretos quando a aplicação estiver operando em um cenário real de ocorrência de falhas. Por essa razão, é fundamental modelar acuradamente a ocorrência de falhas para esses cenários, e assim poder definir uma carga de falhas adequada para um injetor.

Na literatura são apresentados diferentes modelos para emular falhas de comunicação em redes sem fio. Segue a descrição dos modelos para perda de pacotes mais referenciados na literatura [Hohlfeld 2008] e que foram usados como base para a definição da carga de falhas do injetor Mob-FI.

3.1 Modelo de perdas sem memória de Bernoulli

É um dos modelos mais simples. Determina que as perdas de pacotes ocorrem com uma taxa *r*. Cada pacote tem uma probabilidade fixa de ser perdido, independente de estados anteriores. Essa taxa de perda pode ser estimada a partir de uma amostra, dividindo a quantidade de mensagens perdidas pelo total de mensagens enviadas na amostra.

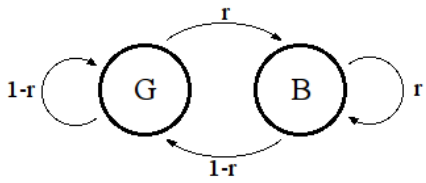


Figura 1: Cadeia de Markov representando o Modelo de Bernoulli

A Figura 1 mostra o modelo de Bernoulli representado com uma Cadeia de Markov de com estados. No estado G (“Good” ou “Gap”) as mensagens são entregues, no estado B (“Bad” ou “Burst”) as mensagens são perdidas. O parâmetro característico desse modelo é a taxa *r*, que representa a probabilidade de ir para o estado de falha e de ficar no estado de falha.

3.2 Modelos de perdas em rajada de Gilbert

O modelo de Gilbert simplificado (Figura 2) utiliza uma Cadeia de Markov semelhante à Figura 1, mas com probabilidades diferentes de transições entre estados. Na Figura 2, *p* define a probabilidade de ocorrer uma rajada de perdas, enquanto *r* afeta o tamanho da rajada. Os valores de *p* e *r* devem ser suficientemente pequenos para simular perdas corretamente [Gilbert 1960].

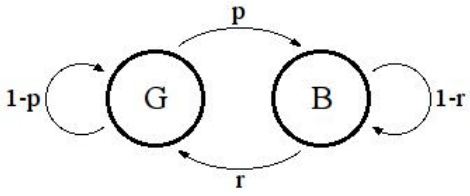


Figura 2: Cadeia de Markov representando o Modelo de Gilbert simplificado

No modelo de perdas completo de Gilbert, além de *p* e *r* é considerado também o parâmetro *h*, que é a probabilidade de não ocorrência de perdas no estado B.

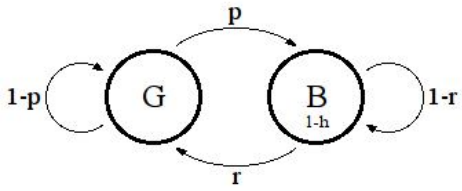


Figura 3: Cadeia de Markov que representa o Modelo de Gilbert

No estado B, Figura 3, a probabilidade de perda de mensagens é de *1 - h*. O estado B é análogo ao Modelo de Bernoulli, com a taxa de perda *r* sendo igual a *1 - h*, podendo este estado ser também modelado com uma Cadeia de Markov com dois estados.

Portanto, no Modelo Completo, a probabilidade final de perda de mensagens será uma combinação do estado do sistema (determinado por *p* e *r*) e a probabilidade *h*.

3.3 Modelo de perdas em rajada de Gilbert-Elliott

No modelo de Gilbert-Elliott [Elliott 1963] é considerada uma taxa de erro também no estado G, porém menor que no estado B. O valor *k* na Figura 4 representa a probabilidade de não ocorrência de perdas.

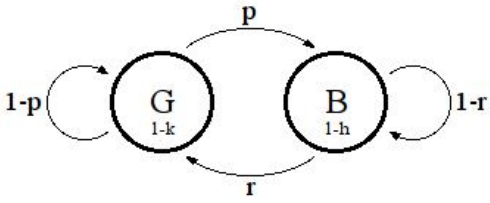


Figura 4: Modelo de Gilbert-Elliott

O modelo permite representar de forma mais satisfatória os dois estados que são característicos de redes sem fio; um estado bom (G) onde ocorrem perdas esporádicas de pacotes, e um estado ruim (B) onde ocorrem perdas em rajadas com alguns poucos pacotes que podem ser transmitidos corretamente.

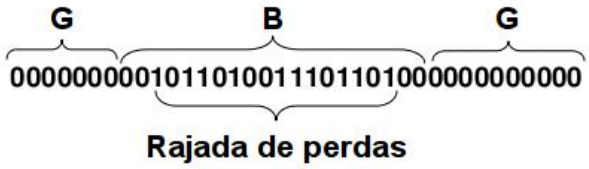


Figura 5: Fluxo de transmissão para h igual a 0,5

A Figura 5 [Salsano 2009] mostra um fluxo de transmissão, onde mensagens transmitidas são simbolizadas com 0 e mensagens perdidas com 1. Para esse fluxo a probabilidade *h* é igual à 0,5, como proposta por Gilbert.

Nota-se que, para *h* maior que 0, a duração do estado B não corresponde a largura da rajada, que é delimitada pela primeira e a última perda de mensagem. Esse comportamento é considerado um ponto fraco do modelo, uma vez que, através da manipulação dos parâmetros *p, r, k* e *h*, não é possível configurar com precisão a largura da rajada.

3.4 Modelo de perdas em rajada com Cadeia de Markov com quatro estados

Salsano (2009) apresenta um modelo que consiste em uma Cadeia de Markov com quatro estados, definida por cinco parâmetros não redundantes.

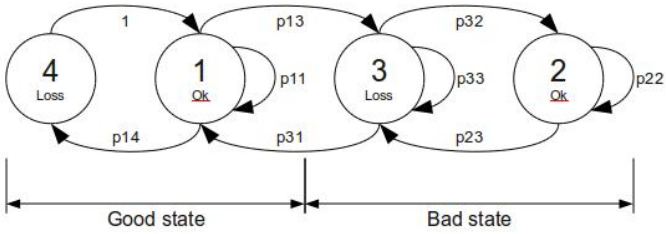


Figura 6: Cadeia de Markov de quatro estados

O modelo é semelhante ao de Gilbert-Elliott, tendo também um estado bom (na Figura 6 marcado como “Good state”) com poucas perdas e um estado ruim (“Bad state”) onde ocorrem algumas rajadas de perdas. Cada um desses estados é dividido em outros dois. Os estados 1 e 2 transmitem mensagens e, nos estados 3 e 4, as mensagens são perdidas. Os parâmetros *p13, p14, p23, p31* e *p32* são suficientes para caracterizar o modelo, sendo os parâmetros *p11, p22* e *p33* calculados como:

$$p11 = 1 - p13 - p14; \quad p22 = 1 - p23; \quad p33 = 1 - p31 - p32$$

No estado bom, existe uma pequena probabilidade *p14* de se passar do estado de transmissão 1 para o estado de perda 4. Para diminuir o número de parâmetros que caracterizam o modelo, a probabilidade de volta do estado 4 para o 1 é igual à 1, que implica em uma única perda por vez. Existe uma probabilidade *p13* de se passar para o estado ruim de rajada de perdas. Como não há transições entre os estados 1 e 2, o estado ruim inicia e finaliza no estado 3, onde efetivamente são perdidas mensagens. Assim, o ponto fraco apontado para o modelo de Gilbert-Elliott não ocorre, e o tamanho da rajada é igual à duração do estado ruim. Esta duração do estado ruim é relacionada à probabilidade *p31* de transição.

A probabilidade *p32* indica a frequência de transmissões durante a rajada de perdas e a probabilidade *p23* a quantidade de transmissões consecutivas dentro da rajada de perdas.

O injetor Mob-FI permite selecionar qualquer um dos cinco modelos apresentados para modelar perda de pacotes. O injetor opera com uma Cadeia de Markov com quatro estados, onde dois são normais e dois são de perda. Os modelos com dois estados são mapeados para os quatro estados manipulando os parâmetros de probabilidade fornecidos pelo usuário na descrição da carga de falhas.

4 Injetor de falhas para o Android

A proposta deste trabalho é injetar falhas de comunicação UDP em aplicações alvo que executam no sistema Android. Como não é possível utilizar ferramentas de intercepção de mensagens disponíveis para outras plataformas, como as Javassist ou JVMTI que já foram usadas por outros injetores de comunicação [Menegotto 2011], [Jacques-Silva 2006], decidimos explorar a solução de modificação do código da máquina virtual.

O injetor Mob-FI, no estágio atual, injeta falhas em mensagens transmitidas e recebidas no protocolo UDP, que é um protocolo de comunicação da pilha TCP-IP muito popular, simples, eficiente, mas não confiável.

4.1 Implementação do injetor Mob-FI

A comunicação UDP em sistemas baseados em Java é realizada pela classe java.net.DatagramSocket. O arquivo DatagramSocket.java contém todos os atributos e métodos da classe. Na implementação do injetor de falhas UDP, foram adicionados atributos e modificados os construtores e os métodos *receive()* e *send()*, que são os métodos para receber e enviar pacotes. O injetor implementa uma cadeia de Markov com quatro estados, onde dois são de transmissão e recepção normal e dois são de perda.

A classe DatagramSocket possui cinco construtores, nos quais foi adicionada ao final uma chamada à função de inicialização do injetor. Nessa função é feita a leitura de um arquivo de configuração e são aplicadas as configurações para o DatagramSocket recém-criado. Caso o arquivo de configuração não esteja completo ou tenha alguma incoerência, o injetor não é habilitado.

O método *receive()* consiste, basicamente, em um laço, onde o método bloqueia até a chegada de um pacote válido para o socket. Quando o pacote for recebido, suas informações são colocadas em uma estrutura do tipo DatagramPacket, que é passada como parâmetro ao método. Para emular a perda de pacotes na recepção, foi adicionado um laço que engloba o laço original da função, onde o estado corrente é verificado e atualizado para o próximo estado. Caso se esteja em um estado de perda, ocorre uma nova iteração, esperando o próximo pacote; caso contrário, a condição de saída do laço do injetor vai ser alcançada e a chamada do método *receive()* é finalizada.

No método *send()*, o pacote passado como argumento é verificado e, caso o pacote seja não válido, é gerada uma exceção correspondente e o pacote não é enviado. Se nenhuma exceção for gerada, o pacote válido é enviado. No método *send()* modificado, os pacotes válidos são enviados apenas se o injetor estiver no estado sem perda de pacotes.

Informações mais detalhadas sobre a construção do injetor Mob-FI podem ser encontradas na documentação sobre a ferramenta [Gindri 2011].

4.2 Execução da carga de falhas

O injetor Mob-FI implementa com uma cadeia de Markov com quatro estados genéricos, de forma a permitir o mapeamento dos cinco modelos apresentados. Os estados são numerados de zero a três, sendo os estados “0” e “1” sem perda de pacotes, e os estados “2” e “3” com perda de pacotes (Figura 7 (a)).

Dentre os atributos da classe DatagramSocket, foi adicionada uma matriz de valores em ponto flutuante de dimensão quatro por quatro, que contém as probabilidades de transição acumuladas, sendo o primeiro índice o estado atual e o segundo índice o próximo estado. A Figura 7 ilustra a cadeia de Markov com todas as transições entre estados e a matriz de probabilidades acumuladas, onde cada linha representa as transições com o mesmo estado de origem e a coluna o estado destino.

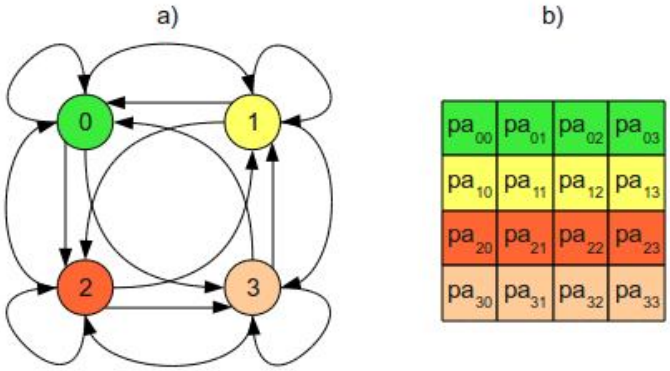


Figura 7: a) representação gráfica. b) matriz de transições.

As probabilidades de transição devem estar contidas no intervalo [0,1], sendo que a soma de todas as probabilidades deve resultar em 1. As probabilidades de transição de cada estado são acumuladas para que um valor escolhido dentro do intervalo [0,1] represente somente uma das transições.

Além da matriz de probabilidades, foi adicionado um atributo inteiro como identificador do estado atual. Também foram adicionados mais dois atributos, do tipo booleano, para indicar se a injeção de falhas será habilitada para o método *send()* e para o *receive()* respectivamente. Apesar do injetor poder ser habilitado simultaneamente para os dois métodos, é utilizada a mesma cadeia de Markov para ambos.

No construtor do DatagramSocket, os atributos que foram adicionados são inicializados a partir de arquivos de configuração. Dentro dos métodos *send()* e *receive()*, se o estado corrente for menor que "2", o pacote é enviado / recebido, caso contrário será perdido. Para calcular o próximo estado, é sorteado um valor randômico no intervalo [0,1), e este valor é comparado com as probabilidades de transição acumuladas na matriz.

4.3 Ajustes do *timeout* para o método *receive*

O *receive()* é, por padrão, um método bloqueante, que espera até ser recebido um novo pacote. Para modificar esse comportamento é possível atribuir um valor de *timeout*, ou seja, um tempo máximo de espera pela recepção de um pacote antes de ser gerada uma exceção.

Originalmente, o controle desse *timeout* é realizado em nível mais baixo que a implementação do DatagramSocket da máquina virtual, causando o reinício da contagem de tempo a cada novo pacote recebido. Isso acontece sempre que uma mensagem for recebida, mesmo que o injetor esteja emulando a perda desse pacote. Entretanto, esse comportamento não corresponde ao que ocorreria no caso de uma perda real de mensagens, pois deveria ser gerada uma exceção de *timeout*. Essa diferença de comportamento é especialmente importante caso o injetor simule uma perda total de conexão, pois isso impediria que a aplicação fosse informada adequadamente dessa perda de conexão. A exceção de *timeout* não seria gerada mesmo que ocorresse uma perda total de conexão emulada, onde todos os pacotes são perdidos.

Para solucionar esse problema foi realizado um ajuste do *timeout*. A classe DatagramSocket modificada mantém o valor de tempo original do *timeout* que foi informado pela aplicação, e este é configurado em mais baixo nível no início de cada

chamada do método *receive()*. Também no início da chamada do *receive()*, é assumido um valor de referência de tempo: cada vez que um pacote é descartado, o tempo transcorrido desde a referência é decrementado do *timeout* original. Caso esse decremento seja maior ou igual ao *timeout* original, é gerada a exceção de *timeout* para a aplicação. Desta forma a falha injetada vai apresentar para a aplicação um comportamento equivalente à falha real.

4.4 Descrição da carga de falhas

Na inicialização, o primeiro arquivo a ser lido é o de configuração inicial, comum a todos os modelos. Através de parâmetros, o arquivo descreve a carga de falhas a ser injetada pelo injetor durante um experimento.

A descrição da carga de falha é composta por quatro números inteiros. Os dois primeiros valores habilitam, caso diferentes de "0", a injeção de falha no envio e recepção, respectivamente. O terceiro valor indica qual modelo será usado, com valores válidos de "0" até "5" representando, respectivamente, os modelos de Bernoulli, Gilbert Simplificado, Gilbert Completo, Gilbert-Elliott, Cadeia de Markov de quatro estados e uma Cadeia de Markov com quatro estados genérica, onde todas as probabilidades de transição podem ser definidas. O quarto valor indica o estado inicial, cujo significado depende do modelo.

5 Avaliação do injetor de falhas para o Android

Para avaliação do injetor de falhas de comunicação Mob-FI foi utilizada uma aplicação simples do tipo Cliente-Servidor como alvo. O Servidor é executado no emulador do Android, e o Cliente roda na máquina virtual no mesmo computador. Para as aplicações Cliente e Servidor poderem se comunicar, é necessário escolher a porta que será utilizada pelo socket UDP e habilitá-la no emulador. Após o emulador iniciar, ele deve ser acessado por telnet no endereço local, na porta do emulador (a porta padrão do emulador é a 5554) e habilitar a porta com o comando de redirecionamento. Neste caso foi utilizada a 3333 para o socket do Servidor. Além de habilitar a porta no emulador, também deve ser habilitada a permissão de acesso a Internet no arquivo AndroidManifest.xml do projeto do Servidor.

O Cliente gera pacotes com um número de sequência e envia para o Servidor. A aplicação envia os pacotes com um tempo de espera suficiente para serem processados pelo Servidor. O tempo foi de 10 milissegundos, e foram enviados 10.000 pacotes.

O Servidor é dividido em duas classes: a aplicação Android principal e uma *thread* de recepção de pacotes. A *thread* de recepção inicia criando o socket UDP e a variável para receber os pacotes. Após, entra em laço, interrompido apenas quando a aplicação for encerrada pelo usuário. Para o teste de avaliação, além de mostrar os resultados no console, o Servidor gera um arquivo, onde cada linha contém o número de pacotes recebidos a cada intervalo de cinquenta pacotes. Para a aplicação conseguir gravar no cartão de memória, foi necessário, além da permissão de acesso a Internet, dar permissão de escrita no seu arquivo AndroidManifest.xml, dentro do projeto da aplicação.

Os testes de avaliação conduzidos comprovaram a capacidade do injetor Mob-FI de emular falhas de acordo com os modelos de perda de mensagens com dois e quatro estados.

5.1 Avaliação do injetor emulando falhas segundo o modelo de Bernoulli

Neste teste, o injetor Mob-FI foi configurado para injetar as falhas na recepção de pacotes de acordo com o modelo para Bernoulli, com a taxa de erro *r* igual a 0,30 (equivalente a 30% de probabilidade de perda de pacote).

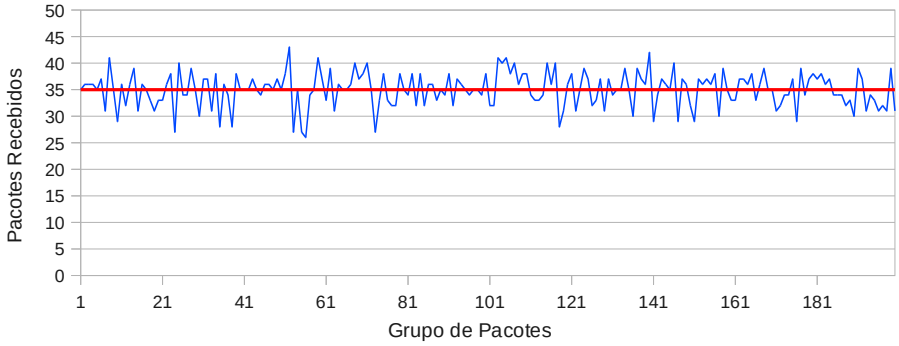


Figura 8: Perdas injetadas de acordo com o modelo de Bernoulli

Na Figura 8, é possível observar a quantidade de pacotes recebidos a cada 50 pacotes consecutivos e, na linha reta, a média de pacotes recebidos durante toda a execução.

Na recepção, os valores estão próximos da média de 35 pacotes, equivalente aos 70% esperados de acordo com a taxa de perda de 30% configurados na carga de falhas para o experimento. O Servidor recebeu 69,99% dos pacotes enviadas pelo Cliente comprovando que o injetor operou corretamente para a carga de falhas descrita.

5.2 Avaliação do injetor emulando falhas segundo os modelos de Gilbert

O injetor foi configurando para o modelo simplificado de Gilbert, com o parâmetro *p* igual a 0,001 e *r* igual a 0,005. Essa carga de falhas corresponde a uma probabilidade de 0,1% de ocorrer uma rajada de perdas de pacote, que deve finalizar com probabilidade de 0,5%.

Como pode ser visto na Figura 9, ocorrem rajadas de erro, onde nenhum pacote é recebido. As rajadas de perda próximas aos grupos de pacotes 70 e 100 não alcançaram o valor zero: são rajadas curtas, pois estão divididas em dois grupos por mensagens que puderam ser recebidas. Nesse teste de avaliação o Servidor recebeu 79,75% dos pacotes enviadas pelo Cliente.

Em um novo teste, agora operando no modelo completo de Gilbert, foi mantido o parâmetro *p*. O parâmetro *r* foi diminuído para aumentar o tamanho da rajada e foi configurada a taxa de envio em estado de erro *h* para 0,20. Com isso, em lugar de não se receber nenhum pacote durante as rajadas de perda, serão recebidos cerca de 20% dos pacotes.

Neste experimento de teste, as rajadas ocorrem de forma semelhante ao experimento anterior (seguindo o modelo simplificado de Gilbert), porém apresentam uma maior duração.

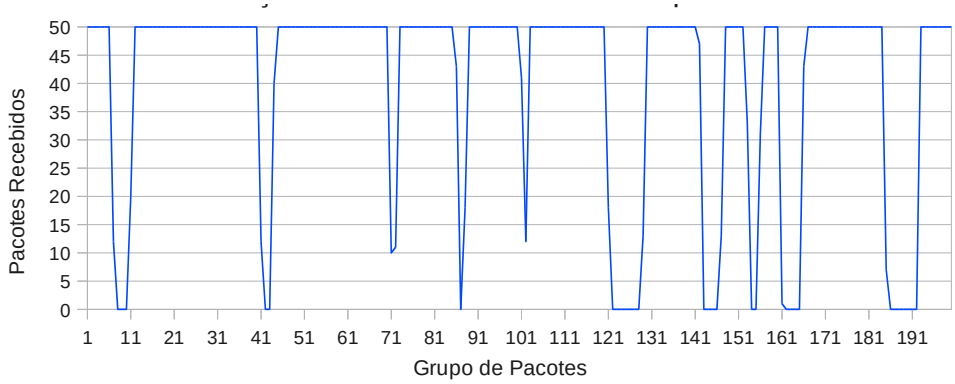


Figura 9: Injeção de perdas seguindo o modelo de Gilbert simplificado

Na figura 10, foi marcada a linha correspondente a 10 pacotes, que equivale a 20% dos 50 pacotes por grupo.

A não ser na rajada de perdas próxima ao grupo 130, que foi a mais curta da execução, as demais rajadas apresentaram um número de pacotes recebidos próximos aos 20% configurados no parâmetro *h*.

Nessa execução do experimento de teste, o Servidor recebeu 86,49% dos pacotes enviadas pelo Cliente.

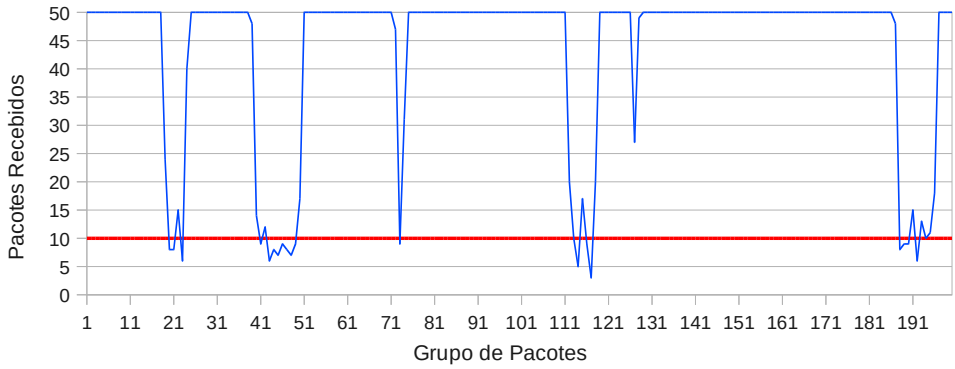


Figura 10: Injeção de perdas seguindo o modelo de Gilbert

5.3 Avaliação do injetor aplicando o modelo de Gilbert-Elliott

O teste realizado com o modelo de Gilbert-Elliott mantém o valor dos parâmetros *p* e *r* dos testes conduzido com os modelos de Gilbert mostrados anteriormente, mas o valor de *h* foi diminuído para 0,1 e o valor do parâmetro *k* foi configurado em 0,85. Com essa configuração, há uma diferença razoável entre as taxas de recepção no estado bom e no estado ruim, facilitando a diferenciação desses estados.

Foram marcadas, na Figura 11, as taxas de recepção esperadas no estado bom, linha reta superior, e do estado ruim, linha reta inferior.

O experimento mostrou que o comportamento da aplicação alvo está coerente com o esperado. Nessa execução do experimento de teste, o Servidor recebeu 70,99% dos pacotes enviadas pelo Cliente.

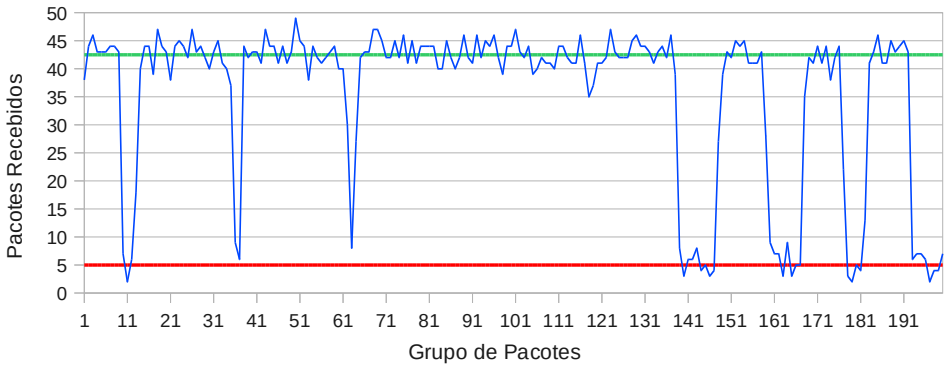


Figura 11: Injeção de perdas seguindo o modelo de Gilbert-Elliot

5.4 Avaliação do injetor aplicando o modelo com quatro estados

Para este teste de avaliação do injetor, os parâmetros do modelo com quatro estados foram configurados de forma a emular rajadas de perda de forma semelhante ao modelo de Gilbert e de Gilbert-Elliot, mas com taxa pequena de perdas no estado bom e uma taxa de transmissão no estado ruim um pouco maior, e de maior duração. Os valores atribuídos aos parâmetros foram *p13* igual a 0,001, *p14* igual a 0,005, *p23* igual a 0,25, *p31* igual a 0,002 e *p32* igual a 0,006.

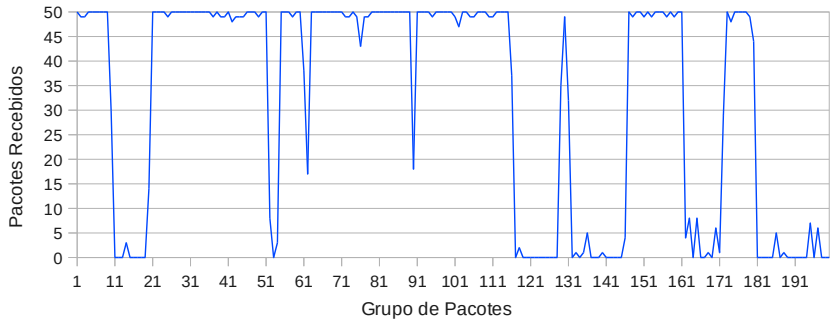


Figura 12: Injeção de perdas seguindo o modelo de quatro estados

No Figura 12 é possível observar os erros simples no estado bom, embora ocorram mais de um dentro de um mesmo grupo em alguns pontos. Também podem ser observados poucos pacotes transmitidos no estado ruim. Com essa configuração ocorreram mais perdas e as rajadas tiveram maior duração. Nessa execução o Servidor recebeu 63,76% dos pacotes enviadas pelo Cliente.

Os testes de avaliação mostraram a capacidade do injetor de emular falhas de acordo com os modelos escolhidos na descrição das cargas de falha. Desta forma, um desenvolvedor ou a equipe de teste pode escolher o modelo de falhas mais apropriado ao teste de sua aplicação móvel, aplicar o injetor Mob-FI em um experimento de teste e com isso determinar o comportamento da aplicação alvo quando sujeita a um cenário de perdas de pacotes. Os cenários emulam situações reais de perdas de pacote em redes sem fio e permitem o teste da aplicação alvo em um ambiente controlado facilitando a observação do comportamento da aplicação alvo sob falhas.

6 Conclusão

O Android, sendo um sistema de código aberto, com vasta documentação e de simples utilização, permite que até mesmo desenvolvedores menos experientes possam criar

aplicações. Somando-se isso com a facilidade de disponibilizar software na web, percebe-se uma oferta muito grande de aplicações vindas das mais diversas fontes, muitas delas desconhecidas e, portanto, pouco confiáveis. Além disso, os dispositivos móveis, como os que executam o sistema Android, estão sujeitos a operar em condições de interferência, atenuação e perda de sinais, o que se traduz em uma grande probabilidade de ocorrência de falhas de comunicação.

Assim, é fundamental dispor de ferramentas de injeção de falhas, que possam ser usadas tanto pelo desenvolvedor e equipe de teste de uma aplicação, que devem verificar a tolerância de suas aplicações a variados cenários de falhas, quanto pelo usuário comum, que tem interesse em saber se o aplicativo que vai empregar será confiável no ambiente real de operação.

Neste artigo, foram relatados os resultados da pesquisa sobre as opções para injetar falhas a partir da máquina virtual do Android. As ferramentas mais comumente usadas para instrumentação nas máquinas virtuais Java convencionais não podem ser utilizadas, devido às diferenças nos bytecodes e na arquitetura da Dalvik Virtual Machine, a máquina virtual do Android. Sendo o código do Android livremente disponibilizado, foi possível explorar a solução de alterar os recursos do núcleo da máquina virtual Java.

Para desenvolver o injetor e testá-lo no emulador de dispositivos, foi necessário o estudo aprofundado de todo o processo de preparação do sistema, obtenção do código fonte, alteração do código e compilação com o SDK. Assim, foi possível criar um injetor de falhas versátil, que consegue emular adequadamente alguns dos modelos mais difundidos de falhas de comunicação do tipo perda de pacotes em rajada. Além de ser flexível, o injetor mostrou baixa interferência no desempenho, o que é importante para garantir que o sistema instrumentado apresente comportamento mais próximo possível daquele de um sistema não instrumentado.

O injetor está disponível para ser empregado em experimentos de injeção de falhas tendo como alvo qualquer aplicação desenvolvida em Java para dispositivos móveis Android, cuja comunicação seja baseada no protocolo UDP. Trabalhos futuros incluem a extensão do injetor para outros protocolos de comunicação e a validação usando o injetor em aplicações que apresentem requisitos explícitos de confiabilidade e disponibilidade.

7 Referências bibliográficas

Acker, E. V. ; Weber, T. S. ; Cechin, S. L. (2010) “Injeção de falhas para validar aplicações em ambientes móveis”. In: Workshop de Testes e Tolerância a Falhas, 11., 2010, Gramado. XI Workshop de Testes e Tolerância a Falhas. v. 1. p. 61-74.

Bornstein, Dan (2008) “Dalvik Virtual Machine Internals”. Google I/O Developer Conference, 2008 - imamu.edu.sa

Butler, M. (2011) “Android: Changing the mobile landscape”. IEEE Pervasive Computing, 2011 - computer.org

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.

De, P.; Anindya Neogi, Tzi-cker Chiueh. (2003) “VirtualWire: A Fault Injection and Analysis Tool for Network Protocols”. 23rd IEEE International Conference on Distributed Computing Systems, pp. 214

Drebes, R. J.; Jacques-Silva, G.; Trindade, J.; Weber, T. S. (2006) “A Kernel based Communication Fault Injector for Dependability Testing of Distributed Systems.” In: First Int. Haifa Verification Conf., Springer-Verlag. v. 3875. p. 177-190.

Elliott, E.O. (1963) "Estimates of Error Rates for Codecs on Burst-Noise Channels", In: Bell System Technical Journal 42, p. 1977-1997

Gilbert, E. N. (1960) “Capacity of a Burst-Noise Channel”. In: Bell System Technical Journal, vol.39.

Gindri, A. F. (2011) “Estudo de Injeção de Falhas para a Máquina Virtual do Sistema Android”. UFRGS. lume.ufrgs.br.

Hoarau, W.; Sebastien Tixeuil, Fabien Vauchelles (2007) “FAIL-FCI: Versatile fault injection” , Future Generation Computer Systems, Volume 23, Issue 7, Pages 913-919.

Hohlfeld, O., Geib, R., Haßlinger, G. (2008) “Packet Loss in Real-Time Services: Markovian Models Generating QoE Impairments”. In: Proceedings of IWQoS, 2008. p. 239-248

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

Jacques-Silva, G. ; Drebes, R. J. ; Gerchman, J. ; Trindade, J. ; Weber, T. S.; Jansch-Pôrto, I. (2006) “A Network-level Distributed Fault Injector for Experimental Validation of Dependable Distributed Systems.” In: 30th Annual Int. Computer Software and Applications Conf., Chicago. IEEE Computer Society Press, 2006. v. 1. p. 421-428.

Looker, N.; Munro, M.; Xu, J. (2005) “A Comparison of Network Level Fault Injection with Code Insertion”. In: Computer Software and Applications Conference, 2005, v. 1. p. 479 – 484

Menegotto, C. C.; Weber, T. S. (2011) “Communication fault injection for multi-protocol Java applications testing.” In: Latin-American Test Workshop, 12th IEEE, 2011, Porto de Galinhas. LATW 2011, 2011. v. 1. p. 1-14.

Salsano, S.; Ludovici, F.; Ordine, A. (2009). “Definition of a general and intuitive loss model for packet networks and its implementation in the Netem module in the Linux kernel”. Technical report, University of Rome “Tor Vergata”, October 2009.

Emulação de Ataques do Tipo *XPath Injection* para Testes de *Web Services* usando Injeção de Falhas

Marcelo I.P. Salas[1], Eliane Martins[1]

[1]Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)
Caixa Postal 676 – 13083-970 – Campinas – SP – Brazil

{marcelopalma@ic.unicamp.br,eliane}@ic.unicamp.br

Abstract. *This paper describes the use of a fault injector to emulate XPath Injection attacks for security testing of Web Services. XPath Injection is a kind of injection attacks, one of the most exploited by attackers, and its consequences are harmful when well succeeded. One of the standards to ensure security in the context of Web Services is the WS-Security (WSS), which, among other mechanisms, uses Security Tokens for access control to messages exchanged between services. The results showed that the use of this mechanism improves the detection of XPath injection, but is still not enough to ensure 100% protection against this type of attack.*

Resumo. *Este artigo descreve o uso de um injetor de falhas para emular ataques de tipo XPath Injection para testar a segurança de Web Services. XPath Injection é um dos ataques de injeção, que são dos mais explorados, além de serem considerados como um dos mais perigosos, quando são bem sucedidos. Um dos padrões para garantir a segurança no contexto de Web Services é o WS-Security (WSS), o qual, entre outros mecanismos, utiliza credenciais de segurança (Security Tokens) para garantir o controle de acesso às mensagens trocadas entre serviços. Os resultados mostraram que o uso desse mecanismo melhora a detecção de XPath Injection, mas ainda não é suficiente para garantir 100% de proteção contra esse tipo de ataque.*

1. Introdução

Os *Web Services* (WS) permitem a interoperabilidade entre sistemas de software desenvolvidos em linguagens de programação diferentes e executados sobre qualquer plataforma [Moorsel et al. 2009]. Utilizando *Web Services*, uma aplicação (cliente) pode invocar outra (servidora), independentemente do local onde resida a aplicação servidora, ou da plataforma em que esta execute. Um WS pode oferecer diversas operações a seus clientes, as quais são descritas usando WSDL (*Web Service Description Language*). Os WS interagem através da troca de mensagens seguindo o protocolo SOAP[1]. Tanto o WSDL quanto as mensagens SOAP utilizam o formato **XML** (*Extensible Markup Language*). O transporte de dados é realizado normalmente via protocolo HTTP (*Hypertext Transfer Protocol*) ou via HTTPS (*Hypertext Transfer Protocol Secure*), no caso de conexões seguras [Hartman et al. 2003].

A realização de negócios e serviços através da Internet faz com que a segurança (*security*) se torne um aspecto ainda mais crítico, de vez que informações importantes

[1] Inicialmente SOAP era abreviatura de *Simple Object Access Protocol*, mas a W3C abandonou tal definição, e hoje em dia SOAP é simplesmente o nome do protocolo, e não mais um acrônimo.

para as empresas circulam através da Internet. Pesquisas realizadas pelo Ponemon Institute e Symantec[2] analisaram os custos devidos a violações de dados de 51 organizações nos EUA em 2010, e revelaram que ataques maliciosos e criminosos se tornaram mais freqüentes, sendo responsáveis por 31% da ocorrência de violação de dados. Além do aumento do número de ataques, o estudo chama a atenção para o alto custo dos mesmos: a diferença entre o custo de violações maliciosas e não maliciosas cresceu mais de dez vezes, ou seja, suas conseqüências se tornam cada vez mais severas.

No caso de *Web Services*, as características que os tornam atrativos, tais como maior acesso a dados e conexões dinâmicas entre aplicações, apresentam novos desafios para a segurança. Além das ameaças tradicionais associadas aos protocolos de rede, tem-se que contar ainda com novas ameaças associadas a novos protocolos e serviços, como SOAP e XML. Um exemplo são os chamados ataques de injeção (*injection flaws*), dos mais explorados em 2010, segundo o relatório anual de segurança do Open Web Application Security Project (OWASP)[3]. Esses ataques também foram considerados como dos mais perigosos, de uma lista de 25 ataques publicados em 2011 por CWE e SANS[4]. Ataques de injeção ocorrem quando dados fornecidos pelo usuário são enviados a um interpretador como parte de um comando ou consulta. Os dados hostis do atacante enganam o interpretador para executar comandos mal intencionados ou manipular dados. A proteção contra ataques desse tipo requer mecanismos de segurança aplicados às mensagens SOAP, para garantir seu transporte até o destinatário final. Foram, para tanto, definidos uma série de padrões para segurança de mensagens, dentre os quais o WS-**Security** (**WSS**) [Holgersson e Soderstrom 2005]. Com base no uso de assinaturas digitais (*XML Signature*), credenciais de segurança (*Security Tokens*) e criptografia de documentos XML (*XML Encription*), WSS visa garantir a segurança na troca de mensagens SOAP.

Uma forma muito comum de determinar se aplicações são vulneráveis a ataques é através do uso de *vulnerability scanners* (VS). Essas ferramentas usam testes de penetração para introduzir ataques e com isso detectar vulnerabilidades. Essas ferramentas diferem em termos do tipo de ataques que elas são capazes de emular. Um estudo recente utilizou várias VS, comerciais e de código aberto, com o objetivo de detectar falhas de segurança em *Web Services* [Vieira et al.2009]; com esse estudo os autores puderam constatar que, para os VS utilizados, houve baixa cobertura das vulnerabilidades conhecidas, e um alto número de falsos positivos. Outra constatação desse estudo é que muitos dos WS em uso são pouco testados em termos de segurança (foram avaliados 300 serviços).

Nossa abordagem usa injetores de falhas para testar a segurança na troca de mensagens entre *Web Services* e seus clientes, ao invés de usar *vulnerability scanners*. O uso de injetores visa obter maior cobertura de ataques, pois permite emular diversos tipos de ataques, por um lado, e também permite variar os parâmetros e dados injetados para emular um determinado ataque. Para reduzir o número de falsos positivos, utilizamos um conjunto de regras, como proposto em [Antunes e Vieira 2009], com base em resultados obtidos de várias fontes.

---

[2] http://www.slideshare.net/symantec/2010-annual-study-us-cost-of-a-data-breach
[3] https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
[4] http://cwe.mitre.org/top25/index.html#Listing

Para mostrar a utilidade da abordagem, foram realizados testes de 10 *Web Services* reais, sendo 5 deles utilizando o padrão WSS e os outros 5, não. Os 10 serviços foram testados em presença de *XPath Injection*, um tipo de ataque de injeção que explora o aplicativo XPath (XML Path Language), ele permite a consulta ou navegação em documentos XML. Os resultados mostraram que o percentual de ataques bem sucedidos passou de 100% a 28% com o uso do WSS.

O texto está organizado da seguinte forma: a Seção 2 apresenta os conceitos e a tecnologia *Web Services*, além dos desafios da Segurança nesse contexto; Seção 3 apresenta a abordagem utilizada. A Seção 4 descreve o estudo experimental realizado e os resultados obtidos estão na Seção 5. A Seção 6 conclui o trabalho, mostrando suas principais contribuições e apontando direções para trabalhos futuros.

2. Aspectos de Segurança em *Web Services*

A segurança é uma qualidade de sistema que garante a ausência de acesso ou manipulação, não autorizados, ao estado do sistema [Avizienis et al. 2004]. A segurança tem como principais atributos a **confidencialidade** (a informação só deve ser revelada para usuários autorizados), a **disponibilidade** (o acesso ao sistema não pode ser negado, de forma maliciosa, a usuários autorizados), e a **integridade** (a informação não pode ser modificada por usuários não autorizados) [Holgersson e Soderstrom 2005].

As violações de segurança dos sistemas ocorrem devido à exploração de vulnerabilidades existentes. **Vulnerabilidades** são falhas (*faults*) introduzidas, intencionalmente ou acidentalmente, durante o desenvolvimento do sistema. Existem inúmeras causas para a existência de vulnerabilidades, dentre as quais podemos citar a complexidade dos sistemas, bem como a falta de mecanismo para verificação das entradas fornecidas. Um **ataque** explora as vulnerabilidades do sistema, de forma maliciosa ou não, podendo comprometer as propriedades de segurança do sistema. O resultado de um ataque bem sucedido é uma **intrusão** no sistema. A Figura 1 ilustra esses conceitos.

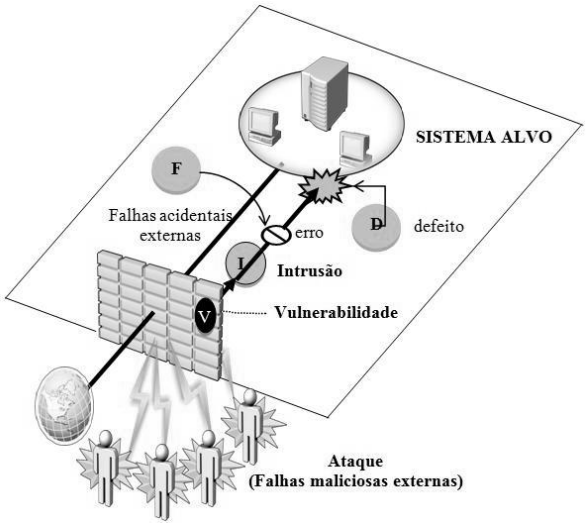


Figura 1. Ameaças à segurança [Cachin e Camenisch 2000].

2.1 Alguns aspectos de segurança em *Web Services*

A segurança em *Web Services* pode ser tratada *ponto-a-ponto* ou *fim-a-fim* [IBM e Microsoft 2011], sendo que diferentes padrões foram propostos para cada contexto. No contexto *ponto-a-ponto* procura-se garantir a segurança no transporte de dados; para isso, existem diversos padrões, dentre os quais o HTTPS, já citado, uma extensão do HTTP. A segurança *ponto-a-ponto* permite garantir a confidencialidade dos dados transportados, mas, no caso em que mensagens passem por WS intermediários antes de atingir o destinatário final, a segurança dessas mensagens não está garantida. A segurança *fim-a-fim* visa proteger a troca de mensagens SOAP entre clientes e servidores. Dentre os padrões propostos podemos citar XML-Signature [Eastlake et al. 2008], que define regras para gerar e validar assinaturas digitais[5] expressas em XML. XML-Encryption [Eastlake et al. 2002], que especifica o processo de encriptação dos dados e sua representação em XML. Ou ainda, Security Token [Lawrence et al. 2006-A], que comprova a identidade do cliente, para que este possa ter acesso aos serviços do servidor, usando credenciais de segurança. Tem-se também a especificação padronizada pelo OASIS, WS-Security (WSS) [Lawrence et al 2006-B], que define um conjunto de extensões ao protocolo SOAP, e utiliza os padrões XML-Signature, XML-Encryption e Security Token, oferecendo: (i) integridade, com o uso de assinaturas digitais para o total ou parte das mensagens, (ii) confidencialidade, permitindo que mensagens SOAP sejam cifradas no todo ou em parte, e (iii) autenticidade, com o uso de credenciais de segurança nas mensagens SOAP.

Além da segurança de mensagens, existem outras dimensões para segurança de aplicações baseadas em *Web Services*, tais como proteção de recursos e políticas de segurança; foram propostos padrões para esses diversos aspectos. Dado que nosso interesse está no WSS, não trataremos desses assuntos aqui. O leitor interessado pode consultar, por exemplo, [Mello et al. 2006] ou [Singhal et al. 2007] para uma introdução mais completa sobre o assunto.

2.2 Técnicas para detectar vulnerabilidades

Para desenvolver *Web services* seguros o provedor tem a sua disposição uma série de ferramentas, linguagens e técnicas. Deve também seguir as boas práticas de segurança, e escolher os padrões de segurança mais adequados. No entanto, é preciso também determinar se o *Web Service* implementado apresenta o nível de segurança desejado. Dado que a segurança pode ser comprometida devido à existência de vulnerabilidades, inúmeras técnicas de detecção de vulnerabiliddes estão disponíveis, devendo ser usadas tanto por provedores quanto por usuários de serviços.

Técnicas estáticas não necessitam execução do serviço ou aplicação em teste; as técnicas mais comuns são a análise estática automatizada e inspeção de código. Já as técnicas dinâmicas necessitam da execução da implementação. Nessa categoria, os Testes de Penetração, Fuzz Testing e Injeção de Falhas (IF) são as mais utilizadas.

Os Testes de Penetração simulam ataques com o intuito de revelar vulnerabilidades. Os Testes de Penetração automatizados são possíveis com o uso de ferramentas denominadas *Vulnerability Scanners* (VS). Existem diversos VS, tanto comerciais (por exemplo, HP Web Inspect, IBM Rational AppScan) quanto de código

[5] As assinaturas digitais visam garantir a integridade e confidencialidade de mensagens [Eastlake 2008].

aberto (e.g. WSDigger e WebScarab). As vulnerabilidades detectadas variam de uma ferramenta para outra. Uma avaliação de várias versões de VS comerciais mostrou que essas ferramentas têm como principais limitações a baixa cobertura das vulnerabilidades existentes e a alta porcentagem de falsos positivos [Antunes e Vieira 2009]. Dado que executamos a IF durante a execução, também utilizamos testes de penetração para detectar vulnerabilidades. À diferença dos testes usando VS, a nossa abordagem permite uma cobertura maior dos ataques possíveis.

Fuzz testing [Miller et al. 1995] é uma técnica que consiste em fornecer entradas inválidas, inesperadas ou aleatórias a um sistema e observar possíveis defeitos como colapso (crash) do cliente ou do servidor, ou lançamento de exceções imprevistas. É uma técnica muito usada para testar a segurança de sistemas computacionais. Como exemplos de trabalhos usando essa técnica para testar a segurança no contexto de *Web Services* podemos citar H-Fuzzing [Zhao et. al. 2009] e a ferramenta SQL Fuzzing [Garcia 2009]. Uma vantagem dos fuzz tester é que são úteis para revelar a presença de falhas mais difíceis de serem reveladas com testes criados manualmente, e que podem ser explorados por um atacante. No entanto, a cobertura de vulnerabilidades conhecidas pode ser baixa. Nossa abordagem permite maior controlabilidade dos ataques gerados.

A injeção de falhas consiste em introduzir, seja por hardware ou por software, falhas ou erros em um sistema e observar o seu comportamento [Arlat et al. 1990]. Existem diversas formas de injetar falhas em um sistema. A mais atrativa, do ponto de vista de custo de realização e facilidade de adaptação a diferentes sistemas e plataformas é a IF por software. Nesse caso, as falhas são introduzidas por um injetor, que é um software responsável por injetar falhas no sistema, seja antes ou durante a execução. Na técnica de IF os testes são constituídos por dois conjuntos de entrada: a carga de trabalho (*workload*) e a carga de falhas (*faultload*). A carga de trabalho representa as entradas usuais do sistema, que servem para ativar suas funcionalidades, enquanto a carga de falhas representa as falhas a serem introduzidas no sistema.

IF foi proposta com o intuito de testar mecanismos de tolerância a falhas, mas já existem inúmeros trabalhos na literatura propondo o seu uso para testar segurança de aplicações. Thomson et al. (2002) testam a segurança de aplicações injetando falhas durante a execução nas chamadas feitas ao sistema operacional. Outro trabalho utiliza um injetor de falhas para testar *firewalls* e sistemas de detecção de intrusão, simulando ataques ao TCP/IP [Wanner e Weber 2003]. [Morais et al. 2009] usam essa técnica para testar um protocolo de segurança usado na comunicação entre dispositivos móveis e a Internet. Para os testes de segurança de *Web Services*, também já existem inúmeros trabalhos, dentre os quais podemos citar os trabalhos [Antunes e Vieira 2009] e [Mello e Silveira 2011], que também utilizam perturbações nas mensagens SOAP para emular ataques, como na nossa proposta. No entanto, os dois primeiros trabalhos utilizam injetores específicos para o tipo de ataque, enquanto nosso injetor é de propósito geral. Além disso, nos demais trabalhos, somente corrupção de mensagens podem ser utilizados, enquanto, no nosso caso, outros tipos de perturbação podem ser realizados, como por exemplo, atraso na entrega de mensagens.

3. Abordagem proposta

Nesta seção começamos por mostrar a arquitetura de teste utilizada, e em seguida, apresentamos passo-a-passo a realização dos experimentos.

3.1. Arquitetura de testes

A abordagem proposta utiliza IF como técnica para testes de segurança. Utilizamos para esse fim um injetor desenvolvido em um trabalho prévio do grupo chamado WSInject [Valenti et al. 2010]. A ferramenta atua como um Proxy entre um cliente e um servidor, e permite injetar falhas de comunicação tanto para os testes de um serviço quanto nos testes da composição de serviços. No presente estudo, o objetivo foi testar serviços isoladamente; nesse caso o injetor intercepta as requisições enviadas pelo cliente, através de mensagens SOAP, antes destas serem repassadas ao servidor, conforme ilustrado na Figura 2.

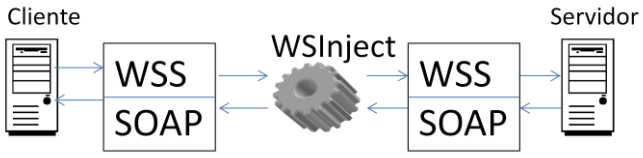


Figura 2. Arquitetura de testes utilizada.

Dado que a WSInject se comporta como um servidor *proxy* HTTP, sua utilização requer pouca instrumentação: basta configurar o cliente para se conectar ao WS alvo através do proxy. A interceptação e modificação das mensagens trocadas entre o cliente e o servidor são transparente para um e outro. Dessa forma, a WSInject não necessita do código fonte do serviço, e nem interfere na plataforma de execução, o que a torna possível de ser utilizada tanto por provedores quanto por usuários do serviço.

A WSInject utiliza scripts para descrição das falhas a serem injetadas [Valenti, Maja, Martins 2010]. Scripts são arquivos de texto contendo um ou mais *FaultInjectionStatements* (comandos de injeção de falhas). Cada *FaultInjectionStatement* é composto de um *ConditionSet* (conjunto de condições) e uma *FaultList* (lista de falhas). Os *FaultInjectionStatements* funcionam como comandos do tipo **condição-ação**: ao interceptar uma mensagem, se esta satisfaz a um conjunto de condições, a lista de falhas é injetada. A Figura 3 mostra um exemplo de script executável pela WSInject. Em negrito temos as palavras-chave para especificar condições e ações. A primeira linha contém uma condição e duas ações, que correspondem às ações que o injetor deve realizar para injetar falhas. Nesse caso, a cada vez que a URI[6] de uma chamada ao Web Service, ou sua resposta, contiver a cadeia "Hotel", substitua todas as ocorrências de "Name" na mensagem por "Age", e duplique o conteúdo da mensagem. Na segunda linha, toda vez que uma mensagem contiver a cadeia "caught exception" e for uma resposta, o conteúdo da mensagem é esvaziado.

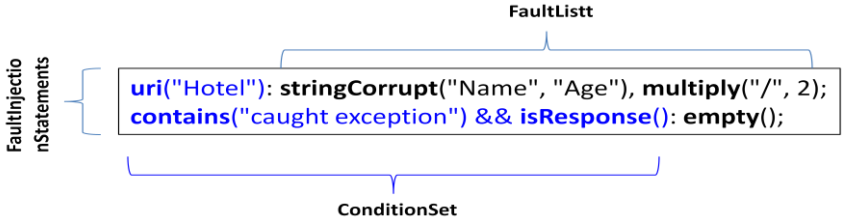


Figura 3. Exemplo de script [Valenti et al. 2010].

[6] Uniform Resource Identifier é uma cadeia de caracteres usada para identificar recursos na Internet.

Uma vez definida a arquitetura de testes, para realizar os ataques foram necessários os seguintes passos:

1. **Preparação**: neste passo foram determinados: Quais ataques realizar? Qual carga de falhas utilizar para emular esses ataques?
2. **Execução**: este passo visa responder às questões: qual carga de trabalho utilizar? Quando injetar as falhas? Que informações coletar durante a execução?
3. **Análise dos resultados**: como determinar se uma vulnerabilidade foi revelada ou não?

Esses passos são explicados mais detalhadamente nas seções a seguir.

3.2.Preparação

Embora tenha sido possível emular diversos tipos de ataques com a WSInject, nesse artigo apresentamos os testes utilizando *XPath Injection*. Conforme mencionado, o atacante, nesse caso, explora comandos XPath (XML Path Language) para atacar servidores que fornecem informação para construir consultas XPath que são usados por usuários para obter dados XML. Através do envio de informações intencionalmente malformadas para o Servidor, um atacante pode conhecer a estrutura dos dados XML e aceder aos dados que são protegidos. O atacante pode até ser capaz de elevar seus privilégios no servidor se os dados XML estão sendo usados para autenticação do cliente. *XPath Injection* pode ter graves consequências, dado que XPath não possui técnicas para controle de acesso e permite a consulta completa dos banco de dados do servidor (documentos XML), permitindo ter acesso até às tabelas de administrador, inacessíveis em consultas regulares [Eviware 2011]. Por exemplo, suponha que o servidor mantenha o arquivo XML mostrado na Figura 4, com informações sobre usuários:

```
<users>
        <user>
         <name>Alice</name>
        <password>ghj348</password>
        <access>Admin</access>
        </user>
        <user>
        ...
</users>
```

Figura 4. Exemplo de trecho de arquivo XML com informações de usuários.

Uma aplicação Web que necessite autenticar um usuário, pede que o cliente forneça seu nome e senha, e gera para o servidor um comando XPath da forma:

```
String xpathQuery = "//user[name/text()='" +
request.get("username") + "' And password/text()='" +
request.get("password") + "']";
```

Um atacante pode fornecer como "*username*" a seguinte cadeia:

```
lol' or 1=1 or 'a'='a
```

Com essa modificação, a autenticação seria ignorada e o atacante conseguiria entrar no sistema.

Para emular esse tipo de ataque, o injetor deve interceptar mensagens SOAP contendo comandos XPath e corromper os valores dos parâmetros. Para definir os valores a serem usados, nos baseamos em informação da literatura, bem como em ataques produzidos por VS, como o soapUI-Security Testing[7]. Exemplos de scripts gerados para esse fim estão mostrados na Tabela 1.

Tabela 1. Exemplos de scripts usados para emular XPathInjection.

Script WSInject	Descrição
`isRequest(): stringCorrupt("<urn:Order Order_ID=\"1\">", "<urn:Order Order_ID=\" or name(//users/LoginID[1]) = 'LoginID' or 'a'='a\">");`	O script usa a condição **isRequest()** que seleciona somente as requisições de um cliente para um servidor. Para cada requisição o injetor realiza **stringCorrupt** para substituir as ocorrências da operação urn:Order Order_ID=\"1\>" por uma consulta XPath (ex.: "<urn:Order Order_ID=\" or name(//users/LoginID[1]) = 'LoginID' or 'a'='a\">") para conhecer a estrutura dos dados XML, e aceder aos dados que são protegidos.
`isRequest(): stringCorrupt("<urn:Order Order_ID=\"1\">", "<urn:Order Order_ID=\"' or '1'='1\">");`	
`isRequest(): stringCorrupt("<urn:Order Order_ID=\"1\">", "<urn:Order Order_ID=\"1/0\">");`	
`isRequest(): stringCorrupt("<urn:Order Order_ID=\"1\">", "<urn:Order Order_ID=\"'%20o/**/r%201/0%20--\">");`	
`isRequest(): stringCorrupt("<urn:Order Order_ID=\"1\">", "<urn:Order Order_ID=\"or name(//users/node()) = 'LoginID' or '1='1\">");`	

O WS-Security oferece proteção contra esse tipo de ataque, usando as credenciais de segurança com informações de autenticação (c.f. §2.1). As informações de segurança são inseridas dentro de *tags* `<wsse:Security>`, sendo que cada mensagem SOAP pode conter um ou mais *tags*.

Dado que uma mensagem SOAP pode passar por vários serviços intermediários até atingir o destino final, o WS-Security permite que os serviços intermediários só possam ler ou modificar os trechos de mensagem que lhes são direcionados. A Figura 5 mostra um exemplo de mensagem SOAP com credencial de segurança.

1 <soapenv:Envelope>	Envelope; toda mensagem SOAP deve conter:
2 xmlns:soapenv="..." xmlns:wsse="...">	Declarações contendo, entre outros, como os dados são representados no documento XML
3 <soapenv:Header>	Início de cabeçalho
4 <**wsse:Security**>	Indica início de trecho seguro
5 <**wsse:UsernameToken** wsu:Id="...">	
6 <**wsse:Username**>Alice</**wsse:Username**>	Credencial que representa identificação do cliente <wsse:UsernameToken>
7 <Password Type="PasswordText">Senha</Password>	
8 </**wsse:UsernameToken**>	
9 </**wsse:Security**>	
10 </soapenv:Header>	
11 <soapenv:Body>	Corpo da mensagem que será transmitida ao WS
12 ...	
13 </soapenv:Body>	
14 </soapenv:Envelope>	Termina o conteúdo da mensagem

Figura 5. Exemplo de mensagem SOAP com credencial de segurança (Security Token).

No caso do exemplo, o WS destinatário é informado que o cliente que envia a requisição foi devidamente autenticado, conforme indicado nas linhas 6 e 7 da Tabela 2.

[7] http://www.soapui.org/Security/

Como parte da fase de Preparação, foram feitos experimentos de pré-análise, que nos permitiram, entre outros: **i)** determinar a carga de falhas a ser injetada, o que incluiu a identificação das operações e dos parâmetros a serem injetados; e **ii)** identificar o comportamento do serviço em presença de falhas que caracteriza um ataque bem sucedido. Os resultados dessa pré-análise serão definidos nas próximas seções.

3.3.Execução

Um aspecto importante no Teste de *Web Services* é a geração do tráfego de rede – a carga de trabalho. A carga de trabalho representa as requisições que ativam o WS alvo. Para ser o mais realista possível durante os testes, dever-se-ia gerar tráfego bem próximo do fluxo real de mensagens SOAP. Para gerar a carga de trabalho (*Workload*) foi utilizada a ferramenta soapUI[8], representando o Cliente mostrada na Figura 2. O tráfego gerado consiste em requisições feitas pelo soapUI aos *Web Services* reais com propósito de emular um cliente real fazendo requisições. Para a campanha de injeção foram selecionados 10 *Web Services*, 5 dos quais usam o padrão WS-Security e 5 deles, não.

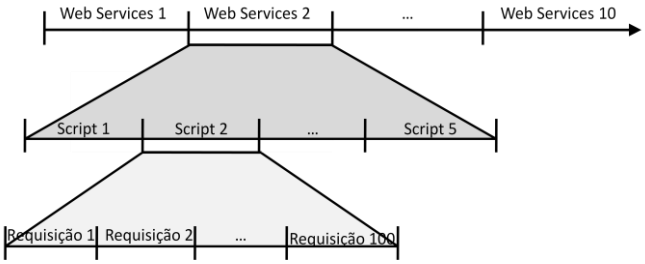


Figura 5. Campanha de Injeção de *XPath Injection*.

A Figura 5 ilustra como foi realizada a campanha de injeções para os serviços testados. Para cada WS foram realizados 5 scripts de injeção, em que cada script especifica a corrupção do valor de um determinado parâmetro, de uma determinada operação, conforme ilustrado na Tabela 1. Para cada script, a carga de trabalho consistiu no envio de 100 requisições. No total, 5000 ataques foram realizados.

Cumpre notar que o número de ataques possíveis de serem realizados, em que cada parâmetro de cada operação de cada serviço seria corrompido com diferentes valores, seria inviável de ser realizado dentro do tempo previsto para a realização dos testes. Por essa razão, optamos por realizar apenas um subconjunto dos testes. Os experimentos de pré-análise nos permitiram determinar as operações e parâmetros que seriam mais interessantes usar como alvo, ou seja, aqueles que levaram a maior ocorrência de defeitos (*failures*) na pré-análise de um conjunto de 69 *Web Services*.

3.4.Análise dos resultados

Um aspecto importante nesse passo é conseguir identificar quando uma vulnerabilidade foi efetivamente detectada, i.e., quando um ataque foi bem sucedido, excluindo potenciais falsos positivos. Devemos diferenciar quando um resultado

[8] http://www.soapui.org/

inválido é obtido devido a uma falha interna (não intencional) do serviço, ou se é consequência de um ataque bem sucedido.

A abordagem proposta é caixa preta, ou seja, não se teve acesso nem ao código fonte dos serviços, e, à exceção de um dos serviços, também não se teve acesso aos respectivos servidores em que residiam. Para determinar se um ataque foi bem sucedido, utilizamos como fontes de informação os logs armazenados pelas ferramentas soapUI e WSInject. Os logs contêm as requisições feitas pelo cliente, bem como a resposta enviada pelo servidor. A figura 6 mostra um exemplo do log produzido pela WSInject, onde estão assinaladas as diferenças tanto na requisição (alterada após a corrupção do parâmetro) quanto da resposta enviada pelo serviço. Como se pode perceber, nesse caso o ataque foi bem sucedido, dado que o serviço não detectou o ataque e respondeu ao cliente.

Script 2
isRequest(): stringCorrupt("<per:PersonID>", "<per:PersonID> ' or '1'='1");

Requisição	Resposta
<soapenv:Body> <per:PersonReq> <per:PersonID> ' or '1'='1Identificador</per:PersonID> </per:PersonReq> </soapenv:Body></soapenv:Envelope>	HTTP/1.1 200 OK // O WS não reconheceu o ataque Content-Type: text/xml;charset=utf-8 Transfer-Encoding: chunked Date: Thu, 08 Mar 2012 18:04:41 GMT <?xml version='1.0' encoding='UTF-8'?> <SOAP-ENV:Envelope xmlns:SOAP-ENV="**http://schemas.xmlsoap.org/soap/envelope/**"> <SOAP-ENV:Body> <PersonRes xmlns:msgns="**http://j2ee.netbeans.org/wsdl/PersonSvc/PersonSvc**" xmlns="**http://xml.netbeans.org/schema/Person**"> <PersonID> ' or '1'='1Identificador</PersonID> <FamilyName>Jose</FamilyName> <GivenName>Martins </GivenName> </PersonRes> </SOAP-ENV:Body> </SOAP-ENV:Envelope>

Figura 6. Exemplo de log gerado pela WSinject.

Além disso foram usados os códigos de status das respostas enviados pelo protocolo HTTP, em especial, o código de erro 500, que indica erros do servidor. Essas respostas foram obtidas analisando-se o log produzido pela WSInject descritos na Tabela 3.

A partir destas informações analisamos as respostas de todos os WS, aplicando um conjunto de regras que nos permitiram identificar quando uma vulnerabilidade foi efetivamente detectada pelos testes, e também quando um ataque foi devidamente rechaçado pelo mecanismo de segurança do WSS, para os serviços que o implementavam. Esse passo é crucial, para reduzir o número de falsos positivos, i.e, para que não fosse indicada uma vulnerabilidade onde não havia. A análise dos resultados se baseou na lista de códigos de status HTTP, descrita na Tabela 4.

Para levantar o conjunto de regras, utilizamos a fase de pré-análise, citada em §3.2, na qual selecionamos um conjunto de 69 serviços. Primeiramente os serviços foram executados sem injeção de falhas, para se determinar as respostas fornecidas. Em seguida, foram introduzidos ataques, utilizando um VS – soapUI-Security Test.

Tabela 3. Lista de Códigos de status HTTP.

Códigos HTTP	Descrição
200 – OK	Padrão de resposta para solicitações HTTP bem sucedida. Vulnerabilidade confirmada porque o sistema executou a requisição sem detectar o ataque.
400 – Bad Request	A requisição não pode ser atendida devido à sintaxe ruim. Consideramos como ataque mal sucedido dado que o servidor detectou o ataque.
500 – Internal Server Error	O servidor não cumpriu com uma solicitação aparentemente válida ou encontrou uma condição inesperada que o impediu de atender a requisição feita pelo cliente. Consideramos analisar a resposta do servidor usando o *tag* <soap:Fault> dentro do corpo (body) da mensagem; essa *tag* fornece os erros e a informação de status da mensagem SOAP, contendo os sub-elementos: • <faultcode> Codigo de identificação da falha. • <faultstring> Explicação legível da falha. • <faulttactor> Informação de quem/que fiz acontecer a falha. • <details> Informação detalhada do erro. Os valores de faulcode podem ser classificados em 4 tipos: • VersionMismatch: O servidor encontrou um espaço de nome (*namespace*) inválido no envelope da mensagem SOAP. • MustUnderstand[9]: A falha de MustUnderstand indica a ausência do um elemento obrigatório no cabeçalho da mensagem SOAP. • Client: A mensagem foi estruturado de forma incorreta ou contém informações incorretas. • Server: Aconteceu um problema com o servidor de forma que a mensagem não possa ser processada.

Com base nos resultados obtidos na fase de pré-análise, e na interpretação dos códigos de status HTTP dado na Tabela 3, criamos um conjunto de regras para analisar os resultados, de forma análoga ao trabalho de Antunes e Vieira (2009), obtendo um conjunto de 11 regras, descritas na Tabela 4.

Os resultados dos ataques para os 10 *Web Services* estão resumidos na Tabela 5. Pode-se notar que todos os ataques especificados puderam ser injetados pela WSInject. De resto, como seria de se esperar, a maioria dos ataques puderam ser detectados pelos serviços que usam WSS. Essa diferença ocorre porque as **credenciais de segurança** verificam a autenticidade do cliente, além de outros parâmetros.

Os 28% de ataques bem sucedidos correspondem a dois serviços que o atacante conseguiu explorar vulnerabilidades no sistema. Um dos WS retornou o código 200 em resposta a requisições corrompidas, ou seja, o serviço não conseguiu detectar o ataque, fornecendo ao atacante a resposta que este esperava (c.f. regra 2 na Tabela 4). O outro reconheceu parcialmente o ataque devolvendo o erro 500 indicando problemas de sintaxes na mensagem SOAP. No entanto, forneceu informações internas detalhadas, que podem servir para outros ataques (c.f. Regra 7 na Tabela 4).

[9] O atributo MustUnderstand serve para indicar se uma entrada no cabeçalho é obrigatório ou opcional.

Tabela 4. Regras para determinar ataques bem ou mal sucedidos em *Web Services*

Regras	Descrição
Regra 1	Se a resposta enviada contém mensagem de tipo "*Ok*" o código "*HTTP/1.1 200 OK*" (ex.: ver o log gerado pelo WSInject na Figura 6), então foi detectada uma vulnerabilidade porque o sistema executa a requisição sem detectar o ataque.
Regra 2	Se a resposta enviada contém mensagem de erro do tipo "bad request message" ou o código "HTTP/1.1 400" (ex.: "Request format is invalid: Missing required soap: Body element."), então não foi detectada vulnerabilidade, pois o servidor foi capaz de detectar a mensagem inválida enviada pelo atacante.
Regra 3	Se a resposta enviada pelo serviço é errônea, tanto na ausência quanto na presença de ataques então não foi detectada vulnerabilidade, ou seja, a resposta errônea não foi causada pelo ataque, mas se deve à existência de outra falha de software.
Regra 4	Se a resposta é válida em presença do ataque e na execução sem falhas houve erro, então foi detectada uma vulnerabilidade, pois o ataque permitiu que mecanismos de autenticação fossem ignorados, ou o acesso a informações sem a devida autorização.
Regra 5	Se a resposta enviada é correta e não houve divulgação de informações sobre execução no servidor (ex.: stack trace), que permita ao atacante tomar conhecimento sobre o software usado no servidor, então não foi detectada vulnerabilidade.
Regra 6	Se a resposta enviada contém exceções geradas pelo analisador XPath, então foi detectada uma vulnerabilidade.
Regra 7	Se a resposta retornada pelo serviço contém informações sensíveis, tais como divulgação do cookie da sessão do usuário (seqüestro de sessão), ou exibe informações detalhadas sobre a conexão do usuário (método de criptografia, o uso de SSL) ou rotas de diretórios do servidor, então foi detectada vulnerabilidade.
Regra 8	Se o servidor retorna código executado na mensagem SOAP ou redireciona o usuário para outra página ou ainda o servidor não responde, então foi detectada vulnerabilidade.
Regra 9	Se a resposta contém exceções geradas pelo servidor de banco de dados, então foi detectada vulnerabilidade, pois o ataque permitiu executar partes do serviço que não o haviam sido quando entradas válidas foram fornecidas.
Regra 10	Se a resposta errônea foi propagada para outras camadas, gerando códigos de erro ou exceções, então foi detectada vulnerabilidade, pela mesma razão dada acima.
Regra 11	Se nenhuma das regras acima pode ser aplicada, então o resultado é tido como inconclusivo, pois não há forma de confirmar se realmente existe uma vulnerabilidade.

Tabela 5. Sumário dos ataques realizados.

***Xpath Injection* – sem WSS**			
	Total	**Ataques Mal Sucedidos**	**Ataques bem Sucedidos**
Total de ataques desejados	2500	0	2500
% Ataques injetados	100%	0%	100%
Total de scripts	25	0	25
***Xpath Injection* com WSS**			
Total de ataques desejados	2500	1800	700
% Ataques injetados	100%	72%	28%
Total de scripts	25	18	7

Tabela 6. Resultados do ataque de *XPath Injection*.

Web Services	**qun**	**min**	**max**	**média**	**tps**	**bytes**	**bps**
Ataque sem WSS	2500	495	3788	715.586	1.329	2432100	1293.2
Ataque com WSS	2500	8	26524	901.682	1.464	2971416	1352.8
Total	**5000**	**8**	**26524**	**808.634**	**1.397**	**5403516**	**1323**

A Tabela 6 resume o desempenho da execução dos scripts, fornecendo as seguintes informações: quantidade de requisições, tempo mínimo de resposta a uma requisição (ms), tempo máximo de resposta a uma requisição (ms), tempo médio de

resposta a uma requisição (ms), transações ou requisição por segundo, quantidade de bytes enviados e número de bytes por segundo. Podemos observar que as requisições feitas a WS com WSS precisam de mais tempo para ser enviadas (média) e enviam maior quantidade de bytes, como já seria de se esperar.

4. Resultados e Trabalhos Futuros

Nesse artigo mostramos uma abordagem para testes de segurança de *Web Services* usando um injetor de falhas para emular ataques. A título de exemplo, mostramos a emulação do ataque *XPath Injection*, em que um atacante modifica consultas a documentos XML feitas com a linguagem Xpath. Trata-se de um ataque bastante frequente, segundo estudos citados, e cujos efeitos podem ser bastante perigosos para o sistema. Os organismos de padronização propuseram a especificação do WS-Security (WSS), que, entre outros mecanismos, usa credenciais de segurança (*security tokens*) para proteger as mensagens trocadas entre serviços contra esse tipo de ataque. Foram testados 10 serviços, sendo que metade com a implementação desse protocolo, e metade não. Os resultados mostraram a eficácia do WSS na proteção contra esse tipo de ataque.

Uma vantagem da abordagem proposta é que ela se baseia no uso de um injetor de falhas de propósito geral, o qual pode ser usado para emular diversos tipos de ataques, inclusive podendo gerar variantes dos mesmos, o que geralmente é limitado nas ferramentas comumente usadas para testar segurança, como os *vulnerability scanners*. Para mostrar a flexibilidade da ferramenta, foram emulados diversos tipos de ataques. Os resultados não foram mostrados aqui por questão de espaço.

Como trabalho futuro, pretendemos utilizar variantes de ataques para melhorar na detecção de novas vulnerabilidades, sempre considerando o serviço como caixa preta.

5. References

Ana, C. V. de Melo; Silveira, P.; “Improving data perturbation testing techniques for Web services”; Inf. Sci. 181, 3 (February 2011), 600-619.

Antunes, N.; Vieira, M.; "Detecting SQL Injection Vulnerabilities in Web Services". Dependable Computing, 2009.LADC '09. Fourth Latin-American Symposium 2009.

Arlat, J.; Aguera, M.; Amat, L.; Crouzet, Y.; Fabre, J-C.; Laprie, J-C, Martins, E. "Fault injection for dependability validation: a methodology and some applications", 1990.

Avizienis, A.; Laprie, J-C.; Randell, B.; “Dependability and Ist Threats: A Taxonomy”; in: Jacquart, R.l eds.; Proceedings of the IFIP 18th World Computer Congress; 2004.

Cachin, C.; Camenisch, J.; “Malicious and Accidental-Fault Tolerance in Internet Applications: Reference Model and Use Cases”, LAAS, MAFTIA, 2000.

Eastlake, D.; Reagle, J.; Imamura, T.; Dillaway, B.; Simon, E; “XML Encryption Syntax and Processing”. W3C Recommendation. 10/Dec/2002.

Eastlake, D.; Reagle, J.; Solo, D.; Hirsch, F.; Roessler, T.; Bartel, M.; Boyer, J.; Fox, B.; LaMacchia, B.; Simon. “XML Signature Syntax and Processing, 2nd Edition”. 2008.

Eviware. soapUI; the Web Services Testing tool – Security Testing Tool.

Garcia, R. "Case study: Experiences on SQL language fuzz testing", DBTest 09 Proceedings of the Second International Workshop on Testing Database Systems.

Hartman, B.; Flinn, D.; Beznosov, K.; Kawamoto, S.. “Mastering Web Services Security”, Wiley Publishing, Inc, ISBN-13: 978-0471267164. Jan. 2003.

Holgersson, J.; Soderstrom, E. “Web Wervice Security-Vulnerabilities and Threats Within the Context of WS-Security”. SIIT 2005, ITU, Geneva, Sep/2005.

IBM Corp.; Microsoft Corp. Whitepaper “Security in a Web Services World A Proposed Architecture and Roadmap”. April 7, 2002, V1.0.

Lawrence, K.; Kaler, C.; Nadalin, A.; Monzillo, R.; Hallam-Baker, P. “Web Services Security: UsernameToken Profile 1.1”. OASIS Standard Specification. 2006-A.

Lawrence, K.; Kaler, C.; Nadalin, A.; Monzillo, R.; Hallam-Baker, P. “Web Services Security: SOAP Message Security 1.1 (WS-Security 2004)”. OASIS, 2006-B.

Mello, E. R.; Wangham, M. S.; Fraga, J. S.; and Camargo, E. S. (2006). Segurança em serviços web. In Minicursos do SBSeg 2006, Santos, SP.

Miller, B. et al.; “Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services”, Computer Sciences Technical Report #1268, 1995.

Moorsel, A. V.; Bondavalli, A.; Pinter, G.; Madeira, H.; Majzik, I.; Durães, J.; Karlsson, J.; Falai, L.; Strigini, L.; Vieira, M.; Vadursi, M.; Lollini, P. Esposito, R. “State of the Art, AMBER - Assessing, Measuring, and Benchmarking Resilience”- report, 2009.

Morais, A.; Martins, E.; “Injeção de Ataques Baseados em Modelo para Teste de Protocolos de Segurança”. Dissertação (Mestrado em Ciências da Computação) – Instituto de Computação, Universidade Estadual de Campinas. 15/Maio/2009.

Morais, A.; Martins, E.; and Cavalli, A. 2009. Security Protocol Testing Using Attack Trees. In Proceedings of 2009 International Conference on Computational Science and Engineering, (Vancouver, Canada, Aug. 2009).

Singhal, A.; Winograd, T.; Scardfone, K.; “Guide to Secure Web Services”; Recommendations of the National Institute of Standards and Technology (NIST).

Valenti, A. W.; Maja, W. Y.; Martins, E.; Bessayah, F.; and Cavalli, A. “WSInject: A Fault Injection Tool for Web Services".Technical Report.ICa10a22, Campinas, "Brazil: Instituto de Computação, Universidade Estadual de Campinas, July 2010.

Vieira, M.; Antunes, N.; Madeira, H. "Using Web Security Scanners to Detect Vulnerabilities in Web Services".Conf. on Dependable Systems and Networks, 2009.

Wanner, P.C.H. and Weber, R.F. 2003. Fault Injection Tool for Network Security Evaluation. LNCS, vol. 2847/2003, Dependable Computing. (Sep. 2003).

Zhao, G.; Zheng, W.; Zhao, J.; Chen, H. "An Heuristic Method for Web-Service Program Security Testing," ChinaGrid Annual Conference, 2009.

Análise do Impacto de Ataques de Poluição Combinado com *Whitewashing* em Sistemas P2P de *Live Streaming*

Rafael Barra de Almeida, Alex Borges Vieira, Ana Paula Couto Silva

[1]Instituto de Ciências Exatas – Departamento de Ciência da Computação
Universidade Federal de Juiz de Fora (UFJF)
Caixa Postal 36.036-900 – Juiz de Fora – MG – Brazil

rafael.barra@ice.ufjf.br,{alex.borges,anapaula.silva}@ufjf.edu.br

Abstract. *P2P live streaming has become a popular method of transmitting live content. Due to its popularity, it may be the target of attacks and opportunistic users behavior. These attacks can lower transmission quality or even make users abandon the system. In this article, we address attacks in a P2P system. More precisely, we analyze a reputation system in a mesh based P2P system under a pollution attack. During the attack, malicious users collude and also change their identities (whitewashing) often trying to cheat the reputation system. We have tested the reputation system on PlanetLab, which confirms it efficiency during an attack without whitewashing. Ours results are twofold: first we show that just verifying package integrity is not an effective protection. In this case, peers have to ask again for the polluted data, generating a significant data transmission overhead. Second, a simple reputation mechanism is effective against pollution attack and peers collusion, but when malicious peers do whitewashing, the mechanism becomes ineffective. In this case, we note up to 180% of overhead due pollution and retransmission.*

Resumo. *Transmissões ao vivo em P2P tornaram-se uma maneira popular para transmissão de eventos ao vivo. Dada essa popularidade, esses sistemas podem ser alvo de ataques e comportamentos oportunistas. Esses ataques podem diminuir a qualidade da transmissão e até mesmo, fazer com que os usuários abandonem o sistema. Neste artigo são tratados ataques a sistemas de transmissão ao vivo em P2P. Mais precisamente, é analisado um sistema de reputação em uma rede P2P baseada em mesh que sofre um ataque de poluição. Durante o ataque, participantes maliciosos conluem e também trocam de identidade (whitewashing) para tentar enganar o sistema de reputação. O sistema de reputação foi testado no PlanetLab e os resultados confirmam sua eficiência em ataques sem whitewashing. Os resultados apresentados se dividem em duas vertentes: primeiro, simplesmente checar os dados e pedir retransmissão não é uma medida eficiente de proteção. Nesse caso, os participantes irão experimentar uma sobrecarga significativa devido às retransmissões. Segundo, o sistema de reputação simples é eficiente contra ataques de poluição e conluio, mas quando há whitewashing, ele se torna ineficiente. Nesse caso, nota-se uma sobrecarga de até 180% devido aos dados poluídos e retransmissões.*

Palavras Chave: P2P Streaming, segurança, ataque, sobrecarga

1. Introdução

Transmissões de vídeo ao vivo na Internet são uma das aplicações mais importantes atualmente [Haizhou et al. 2011]. Grandes emissoras de TV pelo mundo têm investido nesse tipo de aplicação. Por exemplo, a CNN utilizou uma plataforma P2P para auxiliar as redes de distribuição de conteúdo durante a transmissão do discurso de posse do presidente Barack Obama. Esse evento, tido com um dos maiores de transmissão de vídeo na história da Internet, contou com 1.3 milhões de usuários simultâneos dos quais mais da metade dependeu da estrutura P2P [The New York Times 2009].

Tradicionalmente, a arquitetura mais utilizada para distribuição de vídeo é a cliente-servidor. Porém, essa abordagem sofre com a grande necessidade de recursos para lidar com os dados gerados por esse tipo de aplicação. Nesse contexto, a arquitetura P2P (*peer-to-peer* ou ponto-a-ponto) parece ser a mais adequada, pois permite lidar com falhas de rede e aumento da quantidade de usuários sem a necessidade de recursos elevados.

Vários sistemas P2P para transmissão de vídeo ao vivo organizados em malha têm sido implantados com sucesso e são os mais utilizados atualmente [Hei et al. 2007, Hei et al. 2008]. Porém, os participantes desses sistemas se aproveitam de falhas para realizar vários tipos de ataques ao mesmo. Por isso é importante que haja mecanismos que forneçam confiabilidade na transmissão de dados entre os *peers*. Atualmente, um dos mecanismos mais usados para computar confiabilidade é a reputação. Com esse mecanismo os sistemas computam a confiabilidade de maneira distribuída, levando em conta a experiência de cada *peer* e/ou a experiência global do sistema [Marti and Garcia-Molina 2006].

Entre os ataques e comportamentos indesejados se destacam [Seibert et al. 2010]:

1. **Poluição de conteúdo**: quando *peers* maliciosos alteram parte do conteúdo de um vídeo ao transmiti-lo aos seus vizinhos.
2. **Egoísmo (*Free-Riding*)**: quando um *peer* malicioso recebe dados dos seus vizinhos mas não os encaminham para os demais participantes, não contribuindo com o sistema.
3. **Descarte de Dados (*Data dropping*)**: um participante atrai vários parceiros com promessas de servi-los, porém, nega as requisições.
4. ***Whitewashing***: quando um *peer* sai e entra repetidamente no sistema usando uma nova identidade para evitar sofrer as penalidades devido a mal comportamento.

Algumas tentativas foram bem sucedidas ao combater os tipos de ataques citados anteriormente [Borges et al. 2008], [Seibert et al. 2010]. Porém, essas soluções ainda podem falhar quando o nó malicioso faz *whitewashing*. Devido à facilidade de se conseguir nova identificação [Friedman and Resnick 1999] e à grande dificuldade de se identificar um *whitewasher*, esse comportamento se torna difícil de lidar em sistemas P2P baseados em reputação [Chen et al. 2009]. Poucas soluções para esse comportamento são encontradas na literatura. Algumas serão discutidas na seção 2.

Nesse trabalho, analisa-se o impacto de ataques de poluição quando combinado com *whitewashing* em sistemas P2P de streaming. Foi desenvolvido um protótipo de aplicação que foi executado no PlanetLab, o que permitiu uma avaliação do sistema em um ambiente real. O sistema foi testado em três cenários diferentes. Inicialmente foi verificado o comportamento do sistema na presença de participantes maliciosos que

disseminam poluição durante toda a transmissão. Com esse cenário é possível ver os danos causados pelo ataque de poluição ao sistema. No segundo cenário foi inserido o mecanismo de defesa, baseado em reputação local. Nesse cenário, é avaliada a eficiência desse mecanismo em isolar os parceiros maliciosos e conter a disseminação de conteúdo poluído. Em um terceiro cenário os poluidores vão também fazer *whitewashing*. Dessa maneira será possível ver se esse comportamento realmente causa danos ao sistema.

Resultados mostram que com apenas 10% dos participantes maliciosos o sistema chega a apresentar 96% de dados poluídos e podem gerar uma sobrecarga de aproximadamente 230% na rede. Com o mecanismo de defesa ativo a poluição e a sobrecarga são reduzidas a menos de 4%. Porém, quando os participantes maliciosos fazem *whitewashing* o mecanismo de defesa adotado se torna ineficaz. Nesse caso a poluição observada no sistema chega a 84% e a retransmissão gerada a mais de 100% após 1 hora de execução.

O restante desse trabalho é organizado da seguinte forma: a Seção 3 mostra como a análise do sistema foi feita, apresentando o funcionamento do sistema, o mecanismo de defesa utilizado, a forma como é feito o *whitewashing* e os cenários testados. A Seção 5 mostra através de gráficos os valores obtidos dos experimentos.

2. Trabalhos relacionados

Na tentativa de combater ataques de poluição e *whitewashing* em sistemas P2P de *streaming* duas principais abordagens tem sido implementadas:

A primeira faz uso de uma entidade centralizada de confiança, que é responsável por associar fortes identidades ligadas a informações únicas dos *peers* recém-chegados no sistema. Essa abordagem, além de introduzir um ponto único de falha, ainda vai de encontro à natureza descentralizada dos sistemas P2P [Oualha and Roudier 2009]. A segunda abordagem impõe uma penalidade a todos os recém-chegados no sistema. Contudo, foi mostrado em [Feldman et al. 2006] que essa abordagem pode afetar negativamente a escalabilidade do sistema, já que *whitewashing* é um comportamento que não pode ser observado e, sendo assim, a penalidade é aplicada a todos os *peers* recém-chegados. Além disso, o desempenho do sistema também seria reduzido no caso em que houvesse muita rotatividade dos *peers*.

Em [Seibert et al. 2010] os autores detalham alguns dos compromissos feitos implicitamente pelos participantes enquanto trocam dados. Mostram que, quando esses compromissos não são aplicados de forma explícita, eles podem ser explorados por nós maliciosos para conduzir ataques ao sistema. É proposto ainda um mecanismo baseado em reputação que leva em conta a experiência individual de cada participante e o depoimento global do sistema para combater ataques de *data dropping* em sistemas P2P de *streaming*. Por marcar todos os novos nós como suspeitos, reduzindo os recursos disponíveis a esses nós, os autores acreditam que tal mecanismo também desencoraje os participantes a praticar *whitewashing*. Em [Chen et al. 2009] é apresentado um modelo que tenta capturar as diferenças de comportamento entre *whitewashers* e os demais *peers* baseando-se na observação do sistema no tempo. Os autores afirmam que cada *peer* deve manter uma sequência de ações similares mesmo quando saem e entram novamente no sistema. Sendo assim poderiam ser identificados e toda tentativa de conexão com algum outro *peer* do sistema seria negada. Em [Haizhou et al. 2011],

com o apoio de um *crawler* foi feita uma avaliação do impacto da poluição no PPLive levando em consideração a evolução dinâmica dos participantes assistindo a um canal, o tempo de vida dos usuários e a frequência de chegada e saída de *peers* no sistema. Os autores mostraram que um simples poluidor é capaz de comprometer todo sistema. Em [Lin et al. 2010], com o auxílio de um simulador, os autores mostram que o impacto e a efetividade dos ataques de poluição não são dependentes do tamanho da rede, mas sim da estabilidade e da disponibilidade de largura de banda dos poluidores e da origem. Em [Dhungel et al. 2007], é apresentado um experimento, no qual um poluidor é inserido em um sistema real. Os resultados obtidos mostram que ataques de poluição podem comprometer um sistema P2P de transmissão de vídeo ao vivo. Além disso, são sugeridas algumas técnicas possíveis para verificar a integridade do fluxo de mídia distribuído.

3. Metodologia e Cenários de Experimentação

Foi desenvolvido um protótipo de aplicação baseado em sistemas P2P organizados em malha que foi executado no PlanetLab com uma abordagem semelhante a do SopCast[1], onde os nós são organizados em malha e cada um requisita ou envia dados a seus parceiros. Essa abordagem permite ao sistema ser escalável e resistente a falhas [Hei et al. 2008]. Nesse tipo de sistema cada nó faz *broadcast* periodicamente informando a seus vizinhos quais *chunks* possui. Ao receber uma requisição alguns participantes maliciosos podem alterar propositalmente o conteúdo dos *chunks* antes de enviá-los a seus parceiros, fazendo com que estes recebam dados que não correspondem aos requisitados. Dessa maneira se configura o ataque de poluição. Na ocorrência desse tipo de ataque, os nós "honestos", caso não identifiquem esses dados como poluídos, podem encaminhá-los inocentemente aos seus parceiros, contribuindo assim para a disseminação do conteúdo poluído. Esse cenário pode reduzir drasticamente o desempenho do sistema.

Nesse trabalho não é diferenciada a poluição de conteúdo gerada intencionalmente da poluição gerada por acidente, como por exemplo, quando um participante tem problemas de transmissão que corrompem os dados ou quando um pacote chega atrasado, tudo isso é considerado poluição. Com isso é possível abranger mais situações reais de uma rede, ao mesmo tempo em que evita usuários com problemas de rede que comprometem suas transmissões.

O sistema simulado contém um nó especial chamado origem ou servidor, que é o responsável por gerar o *streaming* de vídeo e dividi-lo em *chunks*. Quando um *peer* entra no sistema ele inicialmente contacta uma unidade centralizada chamada *bootstrap*. Este, por sua vez, responde ao recém-chegado com uma lista de participantes que tem interesse no mesmo *streaming* de vídeo, além da posição atual do vídeo na origem. O *peer* recém-chegado irá então tentar se conectar com os *peers* presentes na lista recebida do *bootsrap* para formar sua lista de vizinhos. A partir desse momento o *peer* pode fazer download e upload de acordo com seus interesses.

O desempenho do sistema será medido em função da quantidade de *chunks* poluídos presentes e pela retransmissão gerada pela necessidade de novas requisições ao receber dados poluídos.

[1]Uma das aplicações mais populares de P2P streaming

Foram realizados testes com 120 nós do PlanetLab, todos entrando no sistema ao mesmo tempo. Cada teste durou 60 minutos. O servidor de vídeo e o *bootstrap* estão em uma máquina dedicada na rede do campus da UFJF. Cada participante se conecta a 18 vizinhos escolhidos aleatoriamente a partir da lista recebida do *bootstrap* (chamados de ativos). Caso o tamanho da lista seja maior que o número máximo de vizinhos, os excedentes ficam em uma lista de que pode ser usada caso algum *peer* da lista de ativos falhe. A cada segundo os participantes enviam uma mensagem de verificação para seus vizinhos e para o *bootstrap*. Essa mensagem é usada para indicar que o participante ainda está "vivo" além de carregar o seu mapa de chunks. O *bootstrap* responde a essas mensagens com uma nova lista de *peers* e a posição atual do vídeo na origem.

Foi incluído um marcador no cabeçalho dos pacotes poluídos para que seja possível identificá-los quando são transmitidos através da rede. Em um cenário real, qualquer esquema de verificação de dados proposto (e.g. assinatura baseada em *hash*) [Haridasan and van Renesse 2007] poderia ser usado para identificar automaticamente os *chunks* poluídos. Assim, nesse trabalho, é ignorado qualquer sobrecarga que possa ser gerada por essas de técnicas de verificação, concentrando-se na sobrecarga causada pela retransmissão de *chunks* devido a recepção de conteúdo poluído.

Foi verificado um sistema onde existem 10% de nós maliciosos que disseminam dados poluídos durante todo o tempo de vida, chamaremos este de cenário 1, onde não há nenhum mecanismo de defesa ativo e nem a pratica de *whitewashing* pelos poluidores. Nesse cenário, os poluidores forjam ter todos os *chunks* passando o seu mapa de *chunks* completo para seus parceiros. Quando recebem uma requisição de *chunk* enviam um dado que não condiz com o pedido. Com esse cenário é possível ver o impacto do ataque de poluição no sistema.

Num segundo cenário verificamos o comportamento do sistema quando o mecanismo de defesa está ativo. Todos os participantes que não são poluidores calculam a reputação de seus parceiros usando o modelo de reputação local apresentado em 4. Quando um parceiro é identificado como poluidor ele é removido da lista de vizinhos e toda tentativa de conexão feita por esse parceiro é negada.

Por fim, verificamos as condições do sistema quando os participantes poluidores também praticam *whitewashing* periodicamente. Nesse cenário o sistema de defesa está sempre ativo. Com isso é possível verificar se o do modelo de reputação local ainda é capaz de combater efetivamente o ataque de poluição quando existem poluidores que fazem *whitewashing*. Em todos os casos existem 10% de nós maliciosos que vão poluir durante todo o tempo de vida do sistema.

4. Mecanismo de Defesa: Reputação

Nesse trabalho é utilizado um modelo de reputação local, onde cada participante monitora a troca de dados com cada parceiro para computar sua reputação. O objetivo é permitir que cada participante identifique os parceiros que disseminam conteúdo poluído e possa isolá-los. No modelo de reputação global, os participantes calculam a reputação de seus parceiros baseados em dois componentes: a experiência individual e o depoimento da rede [Borges et al. 2008]. Não há certeza de que usar o depoimento da rede para computar a reputação de um parceiro tem um bom custo-benefício para aplicações P2P de *streaming*. Por algum motivo, em pequenos intervalos de tempo, um participante

pode se comportar de maneiras muito diferentes com diferentes parceiros, enviando dados poluídos para somente alguns deles. Sendo assim, esperar que a rede convirja para uma opinião consistente pode demorar muito. Dado que atualmente as taxas de transmissão ao vivo excedem 5 *chunks* por segundo, um *chunk* danificado ou poluído se espalha muito rápido em uma transmissão P2P ao vivo, impactando assim muito mais na qualidade da *streaming* do que em uma aplicação de compartilhamento de arquivos. Além disso, um participante p_i pode não ter muitos parceiros que compartilham uma parceria com um dado participante p_j. Assim, o depoimento da rede no p_j pode não ser confiável.

Assim, nesse trabalho será usado um mecanismo de reputação mais simples, descentralizado que se baseia somente na experiência individual que cada participante tem com seus parceiros para calcular a reputação. Mais precisamente, em intervalos regulares de tempo um participante p_i computa a reputação de cada parceiro p_j, $R_i[p_j]$, como sua experiência individual com p_j medida naquele intervalo de tempo, que é computado de acordo com a Equação 1. Durante cada intervalo de tempo p_i requer *r chunks* a p_j. O parceiro p_j pode prover *n* respostas ruins para p_i (onde $0 \leq n \leq r$). Aqui uma resposta ruim é qualquer uma que force p_i a pedir o dado novamente para outro parceiro na rede P2P. por exemplo, uma resposta com dado poluído ou uma não resposta de p_j. A razão n/r dá a qualidade da experiência de p_i com p_j.

Se a razão n/r está abaixo do limite T_i^{max}, p_i diminui a reputação local de p_j. Caso contrário, aumenta sua pontuação. A reputação de p_j em p_i, $R_i[p_j]$, é atualizada como:

$$R_i[p_j] = \begin{cases} max(0, R_i[p_j] - \alpha_{p_i} * (1 + n/r)^{y_i}) & \text{se } n/r > \mathrm{T}_i^{max} \\ min(1, R_i[p_j] + \alpha_{g_i} * (1 - n/r)) & \text{caso contrário} \end{cases} \quad (1)$$

onde α_{p_i} e α_{g_i} são fatores de penalidade e recompensa, respectivamente. Foi feito $\alpha_{p_i} \geq \alpha_{g_i}$ para identificar e penalizar rapidamente os poluidores. Dessa maneira, um participante é recompensado linearmente e punido exponencialmente.

Todo participante recém-chegado recebe uma reputação inicial e, além disso, todos os participantes tem uma reputação mínima R_i^{min} ($0 \leq R_i^{min} \leq 1$): se $R_i[p_j]$ cai abaixo de R_i^{min}, p_i remove p_j de sua lista de parceiros.

O principal problema de se basear somente na experiência individual para computar a reputação é que uma vez que o participante p_i considera o parceiro p_j como um poluidor devido a baixa reputação, p_i não terá mais a oportunidade de trocar dados com com p_j novamente. Isto é, do ponto de vista de p_i, p_j será banido e assim nunca será capaz de se reabilitar. No modelo de reputação global, o depoimento da rede poderia ajudar um participante a se reabilitar junto a um velho parceiro. Essa é uma propriedade interessante, já que alguns participantes podem encaminhar conteúdo poluído.

Por isso, é usado um esquema para permitir reabilitação de participantes que se baseia somente na experiência individual. A ideia é mudar dinamicamente a reputação mínima de cada participante p_i em reação às condições da rede, de acordo com o percebido por cada participante. Se p_i sente que a rede está sob ataque, ele aumenta a reputação mínima R_i^{min} assim penalizando seus maus parceiros mais rapidamente. Caso contrário, ele diminui R_i^{min} para permitir parcerias com participantes previamente punidos. Isso é feito independentemente por todos os participantes.

Mais precisamente, são definidos dois estados do sistema, chamados *calmaria* e *tempestade*. O sistema está no estado calmaria na perspectiva do participante p_i, se p_i acredita que não existe poluidor no sistema. Caso contrário, na perspectiva de p_i, o sistema está em tempestade. A cada intervalo de tempo, p_i verifica o estado do sistema e atualiza R_i^{min} de acordo com a Equação 2. Se o sistema está em calmaria p_i diminui seu limite local R_i^{min} por γ_{g_i}; caso contrário R_i^{min} é aumentado por γ_{p_i}. Foi feito $\gamma_{p_i} > \gamma_{g_i}$ de modo a reagir mais rápido a poluidores. os limites RT_i^{min} e RT_i^{max} foram definidos de maneira que $0 \leq RT_i^{min} \leq R_i^{min} \leq RT_i^{max} \leq 1$.

$$R_i^{min} = \begin{cases} max(RT_i^{max}, R_i^{min} + \gamma_{p_i}) & \text{se } \textit{estado tempestade} \\ min(RT_i^{min}, R_i^{min} - \gamma_{g_i}) & \text{se } \textit{estado de calmaria} \end{cases} \quad (2)$$

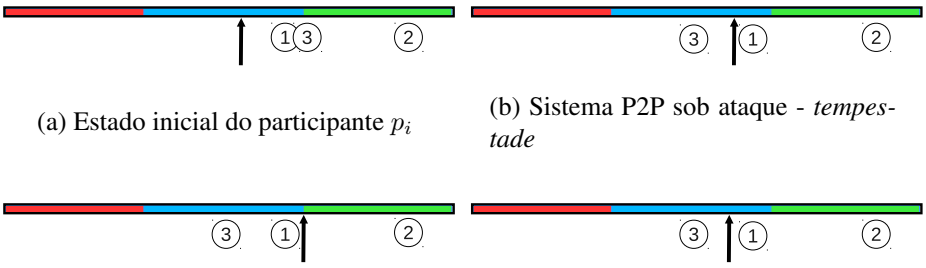
O estado do sistema é definido a partir de cada participante, baseado somente nas suas experiências com seus parceiros. Isto é, se o participante p_i recebe qualquer dado poluído de algum parceiro ele suspeita que o sistema está sob ataque de poluição: sua visão local do estado do sistema é marcada como em tempestade, e ele reage a isso ajustando o limite R_i^{min} de acordo. De maneira similar, se todos os *chunks* recebidos são legítimos, p_i percebe o sistema como em um estado de calmaria e assim reduz R_i^{min}.

A Figura 1 mostra como um participante reage a mudanças no estado do sistema. Na figura, a barra representa a escala de reputação de valores baixos (esquerda) para altos (direita). A seta representa o valor atual de R_i^{min} enquanto que os números 1, 2 e 3, indicam a reputação atual em p_i de três parceiros (referenciados como p_1, p_2 e p_3). Note que, na Figura 4, todos os parceiros estão localizados à direita da seta, indicando que suas reputações estão acima do limite mínimo. Assim que p_i recebe qualquer *chunk* poluído, ele coloca o sistema no estado de tempestade, aumentando seu limite mínimo de reputação, como mostrado na Figura 4. Já que a reputação local de p_3 cai abaixo do mínimo, p_i entende que p_3 é um poluidor, o remove de sua lista de parceiros e pára de interagir com ele. Depois disso, p_i continua recebendo dados poluídos e, novamente, aumenta R_i^{min}, como mostrado na Figura 4. Agora, p_1 também é também é removido da lista de parceiros de p_i, já que sua reputação está abaixo do limite mínimo. Após isso, p_i pára de receber dados poluídos, assim muda o estado do sistema para calmaria e diminui R_i^{min}. O novo valor de R_i^{min} permite p_i aceitar p_1 como parceiro novamente. Note que, mudanças na R_i^{min}, de acordo com a Equação 2, são realizadas independentemente das mudanças nas reputações locais de cada parceiro, definida na Equação 1.

5. Resultados

Os resultados apresentados têm como parâmetros os valores mostrados na Tabela 1. Em todos os experimentos realizados os participantes poluidores permanecem no sistema poluindo durante todo o tempo. Todos os participantes entram no sistema ao mesmo tempo no início do experimento. Para sincronizar essa entrada de nós, foi usado o CRON do Linux. Os resultados foram coletados a cada 5 minutos e os valores apresentados são médias de 5 execuções no Planet Lab de cada cenário.

Inicialmente são apresentados resultados para o cenário onde só existe poluição e nenhum esquema de defesa. Nesse cenário, é possível observar o impacto de ataques de poluição em um sistema real. A Figura 2 apresenta a proporção de dados



(a) Estado inicial do participante p_i

(b) Sistema P2P sob ataque - *tempestade*

(c) Participantes Maliciosos permanecem atacando - *tempestade*

(d) Participantes maliciosos param de atacar - *calmaria*

Figura 1. Limite Dinâmico da Reputação do Participante p_i (seta indica R_i^{min}, 1, 2 e 3 são parceiros de p_i e suas posições na barra indicam suas atuais reputações com p_i).

Tabela 1. Parâmetros de Execução

Parâmetro	Valor
Número de Participantes	120
Tempo de Execução	1 hora
Limite (n/r)	0.5
Reputação Inicial (n/r)	0.7
Reputação Mínima	0.4
Y	2
α_g	0.3
α_p	0.6

poluídos percebidos pelos usuários da rede P2P; e a sobrecarga gerada pelos pedidos de retransmissão. Conforme pode ser observado, com apenas 10% de poluidores, a proporção de dados poluídos no sistema chega a mais de 96%. A retransmissão observada ficou próxima de 230%. Assim, em um sistema sem nenhum mecanismo de defesa, seria necessário mais que o triplo da banda de rede necessária em um cenário sem ataques.

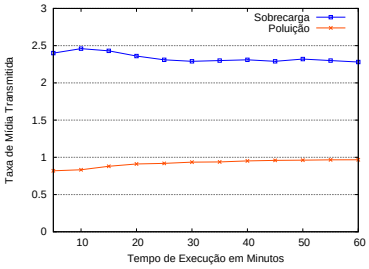


Figura 2. Cenário sem nenhum mecanismo de defesa contra ataques.

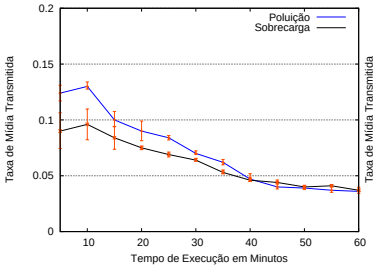


Figura 3. Cenário com reputação implementado e sem ataque de *whitewashing*.

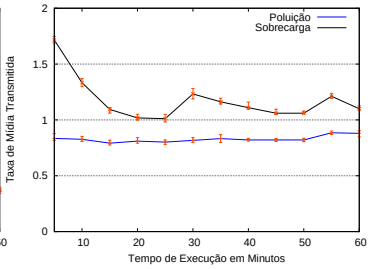


Figura 4. Cenário com reputação implementado e com ataque de *whitewashing*

A Figura 3 apresenta os resultados do segundo cenário implementado, onde o há um mecanismo de defesa. O objetivo desse cenário é verificar se o sistema de reputação é capaz de isolar os participantes maliciosos do sistema, e assim, conter a disseminação de conteúdo poluído. Os resultados encontrados mostram que a proporção de dados poluídos percebidos no sistema P2P e a sobrecarga gerada pelas novas requisições de *chunks* foram

reduzidas a menos de 4%. Mesmo sob influência de variáveis externas como atrasos na rede entre os participantes do PlanetLab, esses resultados demonstram que o sistema de reputação é eficiente sob ataques de poluição e conluio dos atacantes. Neste gráfico a escala foi reduzida para um melhor entendimento do mesmo. Além disso, foi inserido o intervalo de confiança para cada valor para um grau de confiança de 95%. É possível ver que os intervalos são bastante pequenos.

Os resultados do terceiro cenário, onde os poluidores fazem *whitewashing*, são apresentados na Figura 4, juntamente com os intervalos de confiança para cada valor para um grau de confiança de 95%. Nesse cenário, o sistema de reputação sofre com as constantes trocas de identidades. Há uma alta proporção de *chunks* poluídos e retransmissão. Apesar da ineficiência do mecanismo de reputação, com cerca de 110% de sobrecarga em média, ele ainda apresenta resultados consideráveis se comparado a um sistema sem nenhuma proteção.

Concluindo, os resultados mostram que o mecanismo de defesa baseado em reputação local é bastante eficaz ao combater a poluição e conluio no sistema de transmissão ao vivo P2P. Porém, quando os participantes poluidores fazem também *whitewashing* esse mecanismo de defesa se torna ineficaz.

6. Conclusões e Trabalhos Futuros

Este trabalho analisa o impacto de ataques de poluição quando combinado com *whitewashing* em sistemas P2P de streaming. Foi desenvolvido um protótipo de aplicação, executado no PlanetLab, que permite a avaliação do sistema em um ambiente real.

Os resultados mostram que ataques são um problema de fato. A maior parte dos sistemas comerciais não apresenta mecanismos de proteção e nesse caso, um pequeno número de participantes maliciosos podem conseguir um grande efeito em seus ataques. Os experimentos realizados mostram que, com apenas 10% dos participantes maliciosos o sistema de transmissão ao vivo em P2P chega a apresentar 96% de dados poluídos e podem gerar uma retransmissão de aproximadamente 230% na rede. Ou seja, nesse caso, os participantes do sistema P2P devem ter mais de 3 vezes a banda de rede necessária para um sistema sem ataques.

O sistema de reputação analisado consegue mitigar os efeitos de ataque de poluição, mesmo quando há conluio entre os atacantes. Nesse caso, a sobrecarga observada é inferior a 4% da mídia original. Porém, quando os participantes maliciosos realizam *whistewashing*, o mecanismo de defesa torna-se ineficaz. Nesse caso a poluição observada no sistema chega a 84% e a sobrecarga gerada a mais de 100%.

Como trabalho futuro, destaca-se a criação de mecanismos auxiliares para combater os efeitos gerados pelo ataque de *whistewashing*. Nesse caso, deseja-se criar medidas simples e eficientes pois a principal solução adotada na literatura, a criação de mecanismos de identificação confiáveis, por vezes é complicada e custosa. Nesse sentido, serão exploradas maneiras de criar a reputação inicial de um usuário, baseado no seu histórico de parcerias. Essa modificação poderá iniciar participantes confiáveis com boa reputação e dar aos novatos uma reputação mínima. Assim, ao iniciar qualquer tentativa de ataque, os poluidores serão prontamente penalizados e bloqueados do sistema P2P.

Referências

Borges, A., Almeida, J., and Campos, S. (2008). Fighting pollution in p2p live streaming systems. In *Multimedia and Expo, 2008 IEEE International Conference on*.

Chen, J., Lu, H., and Bruda, S. (2009). A solution for whitewashing in p2p systems based on observation preorder. In *2009 International Conference on Networks Security, Wireless Communications and Trusted Computing*, pages 547–550. IEEE.

Dhungel, P., Hei, X., Ross, K., and Saxena, N. (2007). The pollution attack in p2p live video streaming: measurement results and defenses. In *Proceedings of the 2007 workshop on Peer-to-peer streaming and IP-TV*, pages 323–328. ACM.

Feldman, M., Papadimitriou, C., Chuang, J., and Stoica, I. (2006). Free-riding and whitewashing in peer-to-peer systems. *Selected Areas in Communications, IEEE Journal on*, 24(5):1010–1019.

Friedman, E. and Resnick, P. (1999). The social cost of cheap pseudonyms. *Ann Arbor*, 1001:48109–1092.

Haizhou, W., Xingshu, C., and Wenxian, W. (2011). A measurement study of polluting a large-scale p2p iptv system. In *College of Computer Science, Sichuan University, Chengdu 610065, P. R. China*.

Haridasan, M. and van Renesse, R. (2007). Securestream: An intrusion-tolerant protocol for live-streaming dissemination. In *Journal of Computer Communications. Special issue on Foundation of Peer-to-Peer Computing*. Elsevier.

Hei, X., Liang, C., Liang, J., Liu, Y., and Ross, K. (2007). A measurement study of a large-scale p2p iptv system. *Multimedia, IEEE Transactions on*, 9(8):1672–1687.

Hei, X., Liu, Y., and Ross, K. (2008). Iptv over p2p streaming networks: the mesh-pull approach. *Communications Magazine, IEEE*, 46(2):86–92.

Lin, E., de Castro, D., Wang, M., and Aycock, J. (2010). Spoim: A close look at pollution attacks in p2p live streaming. In *Quality of Service (IWQoS), 2010 18th International Workshop on*, pages 1–9. IEEE.

Marti, S. and Garcia-Molina, H. (2006). Taxonomy of trust: Categorizing p2p reputation systems. *Computer Networks*, 50(4):472–484.

Oualha, N. and Roudier, Y. (2009). A game theoretical approach in securing p2p storage against whitewashers. In *Enabling Technologies: Infrastructures for Collaborative Enterprises, 2009. WETICE'09. 18th IEEE International Workshops on*.

Seibert, J., Sun, X., Nita-Rotaru, C., and Rao, S. (2010). Towards securing data delivery in peer-to-peer streaming. In *Communication Systems and Networks (COMSNETS), 2010 Second International Conference on*, pages 1–10. IEEE.

The New York Times (2009). CNN: Inauguration P2P Stream a Success, Despite Backlash.

XIII Workshop de Testes e Tolerância a Falhas

♦

Sessão Técnica 2

Protocolos Distribuídos e Avaliação de Modelos

Exclusão Mútua Distribuída e Robusta para k Recursos Compartilhados

Luiz A. Rodrigues[1,2], Elias P. Duarte Jr.[2] e Luciana Arantes[3]

[1] Colegiado de Ciência da Computação – Universidade Estadual do Oeste do Paraná
Caixa Postal 801 – 85819-110 – Cascavel – PR – Brasil

[2]Departamento de Informática – Universidade Federal do Paraná
Caixa Postal 19.081 – 81531-980 – Curitiba – PR – Brasil

[3]Laboratoire d'Informatique - Université Pierre et Marie Currie
CNRS/INRIA/REGAL – Place Jussieu, 4 – 75005 – Paris, France

luiz.rodrigues@unioeste.br, elias@inf.ufpr.br, luciana.arantes@lip6.fr

Abstract. *This paper presents a robust k-mutual exclusion solution in distributed systems subject to crash failures. The proposed algorithm is based on Raymond's algorithm. To propagate the request messages in a scalable way, we propose a minimum spanning tree algorithm. The tree is created in a distributed manner, based on information provided by an auxiliary monitoring system. The solution improves the efficiency of the Raymond's algorithm in obtaining resources and works correctly for up to n-1 faulty processes.*

Resumo. *Este trabalho apresenta uma solução robusta de k-exclusão mútua em sistemas distribuídos sujeitos a falhas de crash. O algoritmo proposto é baseado no algoritmo de Raymond. Para propagar as mensagens de requisição de forma escalável, foi desenvolvido um algoritmo de árvore geradora mínima. A árvore é criada de forma distribuída, com base nas informações fornecidas por um mecanismo auxiliar de monitoramento de estados dos processos. A solução proposta melhora a eficiência do algoritmo de Raymond na obtenção de recursos e garante o seu funcionamento para até n-1 processos falhos.*

1. Introdução

Um sistema distribuído consiste de um conjunto finito Π de $n \geq 2$ processos independentes $\{p_0, ..., p_{n-1}\}$ que se comunicam usando troca de mensagens, colaborando para a realização de alguma tarefa. Uma das vantagens dos sistemas distribuídos é o compartilhamento de recursos (dispositivos, programas e dados). No entanto, cada processo pode solicitar o acesso a um recurso compartilhado de forma arbitrária. Na programação concorrente, este acesso é normalmente realizado em uma área de código denominada *seção crítica* [Boehm e Adve 2012]. Uma questão relevante é como organizar o acesso concorrente garantindo duas propriedades principais: a segurança (*safety*), que garante que somente um solicitante obtenha o recurso de cada vez e a propriedade de progressão (*liveness*), na qual todos os interessados em um recurso consigam obtê-lo em um tempo finito. A solução para este problema é chamada de *exclusão mútua* [Lamport 1978, Ricart e Agrawala 1981, Raynal e Beeson 1986].

Existem basicamente duas abordagens clássicas para implementar a exclusão mútua em sistemas distribuídos. A primeira é por passagem de permissão (*token*) e

a segunda é através de solicitação de permissão [Raynal 1991]. Com passagem de permissão, somente o processo que detém o *token* pode acessar o recurso compartilhado [Le Lann 1977, Suzuki e Kasami 1985, Raymond 1989b, Naimi et al. 1996]. A permissão pode circular entre os processos seguindo uma organização lógica em anel, por exemplo. Já na solicitação de permissão, cada processo que deseja fazer uso do recurso deve solicitar a todos os demais a permissão para utilizá-lo [Ricart e Agrawala 1981, Sanders 1987]. Para o caso em que um único recurso é compartilhado, a solução trivial é enviar uma mensagem de solicitação a cada um dos outros $n - 1$ processos do sistema e aguardar as respostas [Bertsekas et al. 1991]. Se todos os processos responderem positivamente, o solicitante obtém a permissão. Uma variação da exclusão mútua é a k-exclusão mútua, na qual k recursos são compartilhados entre os n processos [Raymond 1989b, Bulgannawar e Vaidya 1995]. Neste caso, o processo solicitante precisa aguardar, no mínimo, $n - k$ respostas.

Um fator relevante que tem impacto direto na escalabilidade de um algoritmo de exclusão mútua distribuída é o mecanismo de disseminação de mensagens. Uma abordagem simples, empregada na maioria das propostas citadas anteriormente, é utilizar mensagens de *broadcast*. No entanto, em redes nas quais este mecanismo não está disponível, como a Internet, uma maneira eficiente e escalável é empregar uma solução hierárquica, como uma árvore [Avresky 1999].

Outro fator relevante em uma solução distribuída é a possibilidade de ocorrência de falhas. Na exclusão mútua com pedido de permissão, por exemplo, o solicitante precisa ter informações sobre o estado dos processos para não ficar aguardando indefinidamente por respostas daqueles falhos [Bouillaguet et al. 2008]. O mesmo acontece quando se usa passagem de permissão. No caso de falha do processo que possui o *token*, o sistema precisa identificar o problema e gerar um novo *token*. Nas duas situações, uma solução é utilizar um mecanismo de monitoramento que ofereça informações sobre o estado (falho ou sem-falha) dos processos no sistema [Romano e Rodrigues 2009]. Para o problema da exclusão mutua distribuída é necessário um detector perfeito [Delporte-Gallet et al. 2005].

A principal contribuição deste trabalho é uma solução tolerante a falhas de k-exclusão mútua distribuída baseada no modelo com permissão de [Raymond 1989a]. O algoritmo foi adaptado para melhorar a eficiência na obtenção de recursos na presença de até $n - 1$ processos falhos fazendo uso de um mecanismo auxiliar de monitoramento. A segunda contribuição do trabalho é um algoritmo para construção de uma árvore geradora mínima a partir de um nodo fonte qualquer. O algoritmo proposto permite a construção da árvore de forma totalmente distribuída e adaptativa, também com base nas informações de monitoramento e independente da quantidade de processos falhos. Esta árvore é utilizada para propagar as mensagens de requisição, provendo escalabilidade à solução. Os canais são confiáveis, mas processos podem falhar por *crash* e uma falha é permanente. O total de processos é conhecido por todos no início da execução e o sistema é síncrono.

O restante do texto está organizado nas seguintes seções. A Seção 2 discute a exclusão mútua distribuída e o algoritmo de Raymond para k-exclusão mútua. A Seção 3 apresenta o algoritmo de criação da árvore geradora mínima utilizada para a difusão de mensagens. A Seção 4 descreve o algoritmo de k-exclusão mútua proposto neste trabalho. Uma avaliação experimental é apresentada na seção 5. A Seção 6 aborda trabalhos relacionados e Seção 7 apresenta a conclusão e os trabalhos futuros.

2. Exclusão Mútua em Sistemas Distribuídos

A primeira solução para exclusão mútua distribuída baseada em permissão foi apresentada por [Lamport 1978] e utiliza relógios lógicos para determinar a ordem das solicitações. Quando um processo p_i deseja obter acesso ao recurso, envia uma mensagem por *broadcast* para os outros processos e armazena a solicitação em uma fila local. Um processo p_j que recebe uma mensagem de solicitação de p_i, armazena a mensagem na sua fila e retorna uma mensagem de resposta com o *timestamp* atualizado. Assim, p_i pode acessar o recurso quando receber $n-1$ permissões e o seu pedido é o primeiro da fila. Quando libera o recurso, p_i envia uma mensagem *broadcast* de liberação para que os demais processos retirem a solicitação das suas filas, dando oportunidade às demais solicitações pendentes. Cada fase de requisição, resposta e liberação gera $n-1$ mensagens, totalizando $3(n-1)$ mensagens por rodada. O trabalho de [Ricart e Agrawala 1981] aprimorou a solução de Lamport com um algoritmo que requer $2(n-1)$ mensagens. A implementação também utiliza mensagens de requisição e resposta, mas omite as mensagens de liberação. Isto é possível porque quando um processo recebe uma mensagem de solicitação, mas está utilizando o recurso, ele retém a resposta. Assim, quando o processo libera o recurso, ele envia todas as mensagens de resposta adiadas, permitindo que os demais processos tenham a chance de obter o recurso.

Uma extensão do problema da exclusão mútua é a k-exclusão mútua. Nesta categoria, ao invés de um, existem k recursos compartilhados. O objetivo é garantir que, nó máximo, k processos obtenham acesso aos recursos (*safety*) e que todos os processos que solicitarem recursos consigam obtê-lo em um tempo finito (*liveness*). Cada processo pode obter acesso a um único recurso de cada vez. O algoritmo de [Raymond 1989a] soluciona o problema da k-exclusão mútua utilizando uma abordagem de permissões baseada no algoritmo de 1-exclusão mútua de [Ricart e Agrawala 1981]. Quando um processo deseja utilizar o recurso compartilhado, ele envia mensagens de requisição aos $n-1$ outros processos e aguarda por, no mínimo, $n-k$ mensagens de permissão. Se nenhum processo está utilizando ou solicitando recursos, o total de respostas pode chegar a $n-1$. Portanto, no pior caso, são geradas $2(n-1)$ mensagens por solicitação.

A Figura 1 ilustra o comportamento do algoritmo de Raymond em um sistema com 8 processos e 5 recursos. Inicialmente, no tempo t_0 o processo p_0 envia uma mensagem de requisição para todos os demais, solicitando a permissão. Como nenhum deles está utilizando um recurso ou tentando obtê-lo, todos respondem com mensagens de resposta, permitindo que p_0 utilize o recurso em t_1 (o $*$ marca o momento em que p_i obteve o recurso). Em seguida, em t_2 o processo p_1 efetua o pedido de permissão. Da mesma forma, todos os demais processos respondem positivamente, exceto p_0 que está utilizando um recurso. Neste caso, como p_1 deve aguardar por no mínimo $n-k=8-5=3$ respostas, ele também obtém um dos recursos, ainda que p_0 não tenha enviado a resposta. Por fim, quando p_0 libera o recurso, ele envia as respostas referente às solicitações pendentes. Neste exemplo, apenas para p_1 em t_3. Caso p_1 ainda não tivesse obtido respostas suficientes, esta resposta adiada de p_0 poderia completar o total de respostas necessárias.

Intrinsecamente, o algoritmo de Raymond tolera $k-1$ processos falhos. No entanto, cada processo falho degrada a solução, pois pode ser que o número de processos não falhos que não desejam recursos seja insuficiente, fazendo com que o processo solicitante tenha que aguardar um processo sem-falha liberar o recurso e dar a permissão,

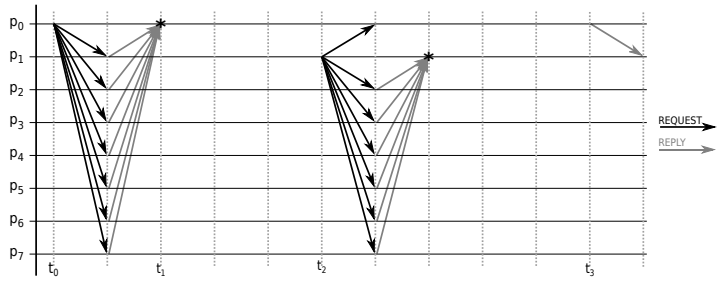


Figura 1. Execução do algoritmo de Raymond para $n = 8$ e $k = 5$ sem falhas.

mesmo que um recurso esteja livre. Voltando ao exemplo anterior, considere que durante a solicitação efetuada por p_1 os processos p_4 a p_7 estejam falhos. O processo p_1 obterá permissão apenas de p_2 e p_3, o que é insuficiente. Neste caso, apenas após a liberação do recurso por p_0 em t_3 é que p_1 conseguirá as três mensagens necessárias para utilizar o recurso, mesmo havendo quatro recursos livres.

3. Monitoramento e Difusão de Mensagens

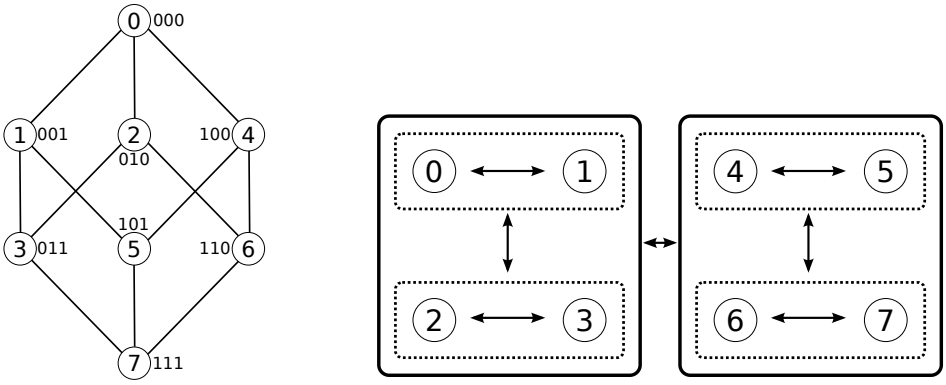
Esta seção está organizada em duas partes. Na primeira é apresentada a estratégia de monitoramento adotada. Na segunda parte é descrito o algoritmo de árvore geradora mínima proposto para a disseminação de mensagens na exclusão mútua.

3.1. Estratégia de Monitoramento

O monitoramento dos processos participantes da exclusão mútua é a base para o funcionamento do mecanismo hierárquico de difusão de mensagens proposto neste trabalho. Além de fazer uso da informação de estado dos processos (falho ou sem-falha), o mecanismo proposto utiliza funções definidas pelo algoritmo de monitoramento Hi-ADSD (*Hierarchical Adaptive Distributed System-Level Diagnosis*) [Duarte Jr. e Nanya 1998].

Considerando um modelo de sistema síncrono, o Hi-ADSD é um algoritmo de diagnóstico distribuído no qual os nodos são interligados logicamente em *clusters* progressivamente maiores, estabelecendo um modelo de testes hierárquico em forma de hipercubo. Um hipercubo de d-dimensões consiste de uma rede com 2^d nodos numerados de 0 a $2^d - 1$. Cada nodo x é identificado pelo código binário $(x_d, ..., x_1)$ do seu identificador. Uma aresta entre dois nodos existe se os seus códigos diferem em um bit, ilustrado pela Figura 2(a).

O modelo de testes do Hi-ADSD é dividido em $\log n$ rodadas, numeradas de 1 a $\log n$. A cada rodada s, um nodo sem-falha testa um nodo no $cluster_s$ e obtém informações sobre os demais nodos naquele $cluster$. Como ilustrado pela Figura 2(b), no primeiro intervalo de testes ($s = 1$), cada nodo testa o *cluster* com um nodo. No segundo intervalo de testes ($s = 2$), o *cluster* com dois nodos. No terceiro intervalo ($s = 3$), com quatro nodos, e assim sucessivamente até que o *cluster* tenha $n/2$ nodos. Em seguida, após mais um intervalo determinado, o processo reinicia. No caso em que todos os nodos são sem-falha, o número de testes executado por cada nodo é $\log n$. A latência de diagnóstico, isto é, o tempo necessário para que todos os nodos sem-falha identifiquem todos os nodos falhos varia entre $\log n$ e $\log^2 n$, no pior caso.



(a) Hipercubo de 3 dim. (b) Organização hierárquica dos testes no Hi-ADSD

Figura 2. Um hipercubo de 3 dimensões e a hierarquia de testes no Hi-ADSD.

Em cada rodada um nodo sem-falha testa o próximo nodo no $cluster_s$ até encontrar um nodo sem-falha ou até testar todos os nodos do *cluster* como falhos. O cálculo dos nodos integrantes de cada *cluster* s e a ordem em que eles são testados por um nodo i qualquer são dados pela função $C_{i,s}$, para $i = 0, .., n-1$. Esta função é definida como:

$$C_{i,s} = < i \oplus 2^{s-1}, C_{i \oplus 2^{s-1},s-1}, C_{i \oplus 2^{s-1},s-2}, ..., C_{i \oplus 2^{s-1},1} >$$

Como exemplo, o *cluster* 3 de n_0 é dado por:

$$C_{0,3} = < 4, C_{4,2}, C_{4,1} > = < 4, 6, C_{6,1}, 5 > = < 4, 6, 7, 5 >$$

3.2. Algoritmo de Árvore Geradora Mínima

Seja $G = (V, E)$ um grafo conexo e não-direcionado com V vértices e E arestas. Uma árvore geradora (*spanning tree*) é um grafo conexo e acíclico que contém todos os vértices de G. Se as arestas possuem pesos, uma *árvore geradora mínima* é aquela cujo a soma dos pesos das arestas é mínima. Se cada aresta possui um peso diferente, existe uma única árvore mínima. Se todas as arestas possuem o mesmo peso, todas as árvores do grafo são mínimas. Em grafos, os dois algoritmos clássicos para a obtenção de árvores geradoras mínimas são o algoritmo de [Kruskal Jr. 1956] e o proposto por [Prim 1957].

O algoritmo de Kruskal inicialmente cria uma *floresta* na qual cada vértice é uma árvore. A cada passo, as árvores são conectadas entre si através das arestas de menor peso. As arestas que não interligam duas árvores são descartas, evitando ciclos. Ao final, uma única componente conexa é gerada e esta constitui a árvore geradora mínima do grafo. O algoritmo de Prim utiliza uma abordagem diferente, que emprega cortes mínimos para escolher as arestas de menor peso e incluí-las na árvore. Além destas soluções centralizadas, algumas propostas foram definidas para soluções distribuídas, baseadas em troca de mensagens. A primeira delas foi definida por [Gallager et al. 1983]. O processo é semelhante ao utilizado por Kruskal. Inicialmente cada nodo é uma árvore. A cada nível, um nodo é eleito líder e uma aresta de peso mínimo que o interliga a um nodo em outra árvore é adicionada. O processo é repetido até formar uma única componente conexa.

O algoritmo proposto neste trabalho dispensa os cálculos dos algoritmos descritos anteriormente, pois faz uso da topologia em hipercubo utilizada pelo Hi-ADSD. Trata-se de um algoritmo distribuído que propaga as mensagens da aplicação através de uma

árvore geradora mínima, construída com base na estrutura hierárquica de *clusters* obtida pela função $C_{i,s}$. O Algoritmo 1 apresenta o pseudocódigo da solução. Ele está dividido em duas tarefas principais: a primeira é realizada pelo nodo inicial que deseja propagar uma mensagem e a segunda pelos demais nodos no hipercubo.

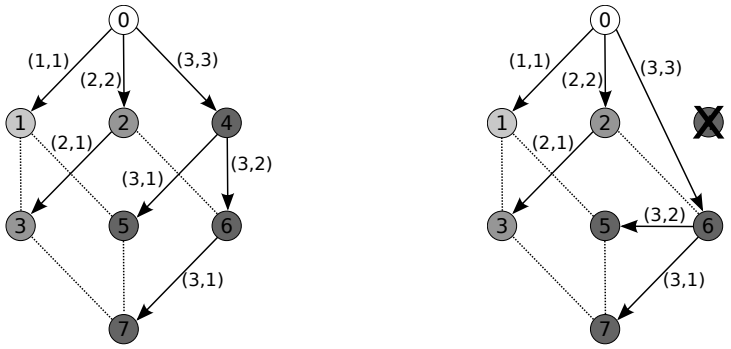
Algoritmo 1 STA(msg): Algoritmo de disseminação das mensagens

```
// Inicialização
 1: req_queue_i ← ∅ // fila de mensagens de requisição pendentes
 2: ∀j ∈ N : last_msg_i[j] ← ⊥ // última mensagem recebida de cada processo

|| ao receber uma nova requisição msg =REQUEST(i, last_i)
 3: req_queue_i ← req_queue ∪ {msg}

|| Tarefa 1: executada constantemente
 4: obtém próxima msg da fila req_queue_i
 5: repita
 6:     para s = 1 até log n faça
 7:         se ∃j ∈ C_{i,s} sem-falha então
 8:             enviar TREE(i, cluster = s, hops = s, msg) para n_j
 9:     aguardar ACK(j, m) ∀j ≠ i, n_j ∉ FD_i.suspects ou timeout
10: até que receba ACK(j, m) ∀j ≠ i, n_j ∉ FD_i.suspects

|| Tarefa 2: executada pelos demais nodos n_i
11: ao receber TREE(r, cluster, hops, msg) de algum n_j
12: se last_msg_i[j] ≠ msg então
13:     last_msg_i[j] ← msg
14:     entregar msg para n_i
15:     enviar ACK(i, msg) para n_r
16: enquanto (hops > 1) faça
17:     hops ← hops − 1
18:     se ∃k ∈ C_{i,s=hops} sem-falha então
19:         enviar TREE(r, cluster, hops, msg) para n_k
```

Considere inicialmente uma execução sem falhas. No primeiro passo, o nodo solicitante envia uma mensagem TREE para cada um dos seus *clusters* (linhas 6-8), totalizando $\log n$ mensagens. Cada mensagem carrega o identificador do solicitante, o *cluster* destino e a quantidade de saltos (*hops*) que a mensagem deverá percorrer. Como cada *cluster* possui uma quantidade de nodos progressivamente maior, quanto maior o seu identificador, maior a quantidade de saltos. Ao receber uma mensagem, o nodo verifica o total de saltos restantes e, se após decrementá-lo, $hops > 1$, encaminha a mensagem para os *clusters* internos subsequentes, iniciando pelos de maior valor. Para cada mensagem enviada, a quantidade de saltos é decrementada em uma unidade (linha 17).

Como exemplo, considere o hipercubo de três dimensões da Figura 3(a). O nodo n_0 inicia o processo enviando uma mensagem para cada um dos *clusters* $C_{0,1} = <1>$, $C_{0,2} = <2,3>$ e $C_{0,3} = <4,6,7,5>$. O nodo n_1 recebe a mensagem, verifica que $hops = 1$ e não a repassa. O nodo n_2 verifica que $hops = 2$, efetua o decremento e repassa a mensagem para n_3, que é o nodo do seu *cluster* $C_{2,1}$. Quando n_3 recebe a mensagem não faz o repasse, já que $hops = 1$. Por fim, o nodo n_4 recebe a mensagem com $hops = 3$ e a repassa para os seus *clusters* $C_{4,2} = <6,7>$ com $hops = 2$ e $C_{4,1} = <5>$ com $hops = 1$. A última mensagem é enviada de n_6 para n_7. Com isso, todos os nodos recebem uma única cópia da mensagem enviada inicialmente por n_0.

Para tratar os casos com falhas, antes de enviar um mensagem o nodo verifica



(a) sem-falhas (b) nodo 4 falho

Figura 3. Árvore geradora no hipercubo de 3 dimensões.

no vetor de estados se existe um nodo sem-falha no *cluster* destino (linhas 7 e 18). Se todos os nodos do *cluster* estão falhos, $hops$ é decrementado, mas a mensagem não é enviada, passando para a próxima tentativa no *cluster* subsequente. A Figura 3(b) ilustra um exemplo com falha. Como o nodo n_4 está falho, a mensagem inicial é enviada para o n_6 com $hops = 3$, que é o próximo no *cluster* $C_{0,3}$ de n_0. O valor de $hops$ é decrementado em n_6. No entanto, como n_4 pertence ao *cluster* $C_{6,2} = < 4, 5 >$ de n_6 e n_4 está falho, n_6 envia o par $(3, 2)$ para n_5. n_5 recebe a mensagem, decrementa $hops$ e verifica que não existe nodo sem-falha no seu *cluster* $C_{5,1}$ e, portanto, não repassa a mensagem. Na última etapa, $hops$ é decrementado em n_6 e o par $(3, 1)$ é enviado para n_7.

O parágrafo anterior discutiu a solução para o caso de falhas detectadas antes do início da disseminação. No entanto, é possível que uma falha ocorra durante o envio das mensagens. Considere, por exemplo, o caso da Figura 3(a). No primeiro passo, n_0 envie a mensagem para n_1, n_2 e n_4. Se n_4 falha após o envio por n_0, a mensagem não será repassada para n_5,n_6 e, consequentemente, para n_7. Esta perda de mensagem, além de reduzir a eficiência, pode levar o algoritmo de exclusão mútua a um estado de interblocagem (*deadlock*). Para garantir que todos os processos recebam a mensagem de requisição, uma mensagem ACK de confirmação é enviada diretamente para o mecanismo de disseminação do processo solicitante (linha 15). Assim, se após um intervalo de tempo determinado não forem recebidos os ACKs de todos os processos sem-falha (espera ativa da condição na linha 9), a disseminação é refeita. Para evitar mensagens de reconhecimento e resposta duplicadas, cada processo mantém localmente um vetor de informações com o *timestamp* da última mensagem recebida de cada processo que solicitou um recurso. Mensagens recebidas com *timestamp* diferente são reconhecidas e entregues à aplicação (condição da linha 12) e as repetidas são apenas repassadas na árvore.

4. Algoritmo de k-Exclusão Mútua Proposto

O algoritmo de k-exclusão mútua proposto neste trabalho é uma adaptação do algoritmo de [Raymond 1989a]. O objetivo das modificações é aumentar a eficiência na obtenção de recursos em cenários com falhas de processos. Assim como na versão original, o algoritmo faz uso apenas de mensagens de requisição (REQUEST) e resposta (REPLY) para solicitar e dar permissão, respectivamente. Para tratar os casos de falha, o solução em questão faz uso de um mecanismo auxiliar de monitoramento distribuído, que informa

o estado (falho ou sem-falha) de cada processo no sistema. O Algoritmo 2 ilustra o pseudocódigo do algoritmo proposto. As variáveis locais mantidas pelos processos são:

- $state_i$: armazena o estado atual do processo, que pode ser $not_requesting$, $requesting$ ou $executing$;
- $clock_i$: usada como relógio lógico local. Inicialmente em zero, é atualizado sempre que uma mensagem de requisição é recebida, passando a armazenar o maior valor entre o relógio local e o *timestamp* da mensagem recebida;
- $last_i$: o valor de *timestamp* da última mensagem de requisição enviada;
- $perm_count_i$: o total de permissões recebidas (mensagens de REPLY) desde a última tentativa de obtenção do recurso;
- $reply_count_i[n]$: vetor que armazena a quantidade de respostas esperadas de cada processo;
- $defer_count_i[n]$: contador que armazena a quantidade de respostas adiadas para cada processo. Após a liberação do recurso, o processo envia todas as permissões pendentes baseando-se nestas informações.

O algoritmo possui duas funções principais: uma função $Request()$ para iniciar o processo de requisição do recurso e uma função $Release()$ para quando deseja liberar o recurso obtido. O processo de requisição inicia com a mudança de estado do processo para $requesting$ (linha 7). Esta mudança garante que se p_i receber um pedido de permissão de um outro processo p_j, ela só será dada caso o relógio de p_j tenha um valor maior que o de p_i ou, em caso de empate, que o identificador de p_j seja menor (linha 21). Em seguida, uma mensagem marcada com o relógio local de p_i é enviada para os demais processos utilizando o algoritmo de *spanning tree* apresentado na Seção 3.

Originalmente, para obter acesso ao recurso é preciso que p_i obtenha permissão de $n - k$ processos. Na proposta em questão, considerando que processos podem falhar, foi adicionada uma espera ativa que consulta o algoritmo de monitoramento para verificar dinamicamente a quantidade de processos falhos (linha 12). Este valor é descontado do total de respostas esperadas, permitindo que o processo obtenha acesso aos recursos com mais eficiência. Ao receber a quantidade de permissões mínima, p_i obtém acesso a recurso e é colocado no estado $executing$. Quando uma falha é detectada durante o $requesting$, o algoritmo verifica se já recebeu a permissão do processo falho (linhas 29-30). Se já recebeu, $perm_count$ é decrementada para não interferir na condição da linha 12, garantindo a propriedade de *safety* (embora o código do monitor não seja apresentado, é preciso garantir que a sua lista de suspeitos seja atualizada apenas após a execução deste ajuste). A liberação de um recurso por um processo p_i implica na mudança de estado do processo para $not_requesting$ e no envio de todas as mensagens de requisição recebidas de outros processos que foram retidas e contabilizadas na estrutura $defer_count$ (linhas 14-18). Com isso, os processos que estão aguardando permissões podem verificar a condição mínima da linha 12 e obter o recurso.

Para garantir o correto funcionamento do algoritmo, duas propriedades precisam ser satisfeitas. Em primeiro lugar, em cada instante de tempo, no máximo k processos diferentes podem estar utilizando os k recursos existentes, caracterizada pela propriedade de *safety* e, em segundo lugar, mas não menos importante, é preciso garantir que se um processo correto solicita um recurso, ele o obterá em um intervalo de tempo finito, de acordo com a propriedade de *liveness*. A prova formal será omitida devido a limitação de espaço, mas as duas propriedades são garantidas.

Algoritmo 2 Algoritmo de k-exclusão mútua robusta

// Inicialização

1: $state_i \leftarrow not_requesting$
2: $\forall j \in N : reply_count_i[j] \leftarrow 0$
3: $\forall j \in N : defer_count_i[j] \leftarrow 0$
4: $perm_count_i \leftarrow 0$
5: $clock_i \leftarrow 0$
6: $last_i \leftarrow 0$

Request(): n_i quer obter um recurso livre

7: $state_i \leftarrow requesting$
8: $last_i \leftarrow clock_i + 1$
9: $perm_count_i \leftarrow 0$
10: STA(REQUEST(i, $last_i$))
11: $\forall j \neq i, j \notin FD_i.suspects : reply_count_i[j] + +$
12: aguardar até que ($perm_count_i \geq N - |FD_i.suspects| - k$)
13: $state_i \leftarrow executing$

Release(): n_i vai liberar o recurso

14: $state_i \leftarrow not_requesting$
15: **para todo** ($j \neq i : j \notin FD_i.suspects$) **faça**
16: **se** ($defer_count_i[j] \neq 0$) **então**
17: enviar REPLY($defer_count_i[j]$) para n_j
18: $defer_count_i[j] \leftarrow 0$

|| ao receber REQUEST(j, $last_j$) de n_j

19: $clock_i \leftarrow \max(clock_i,\ last_j)$
20: **se** ($n_j \notin FD_i.suspects$) **então**
21: **se** ($state_i = executing$ **ou** ($state_i = requesting$ **e** $(last_i,\ i) < (last_j,\ j)$)) **então**
22: $defer_count_i[j] + +$
23: **senão**
24: enviar REPLY(1) para n_j

|| ao receber REPLY($count$) de n_j

25: **se** ($j \notin FD_i.suspects$) **então**
26: $reply_count_i[j] \leftarrow reply_count_i[j] - count$
27: **se** ($state_i = requesting$ **e** $reply_count_i[j] = 0$) **então**
28: $perm_count_i + +$

|| ao receber CRASH(j) de FD_i

29: **se** ($state_i = requesting$ **e** $reply_count_i[j] = 0$) **então**
30: $perm_count_i - -$

Em relação a propriedade de *safety*, considere que, em um dado instante de tempo, k processos estejam utilizando os k recursos disponíveis. Em seguida, um processo p_i inicia a operação de REQUEST, enviando mensagens para os demais processos. Cada processo p_j ao receber a solicitação pode tomar uma das seguintes decisões:

- Se p_j está no estado *not_requesting* ele responde imediatamente com um REPLY para p_i;
- Se p_j está no estado *executing* ele retém a resposta até que libere o recurso e, só então, envia REPLY para p_i;
- Se p_j está no estado *requesting* e $(last_i, i) < (last_j, j)$, ele envia imediatamente REPLY para p_i, pois neste caso p_i tem prioridade. Caso contrário, retém a resposta até que ele consiga obter o recurso e, só depois de liberá-lo, envia REPLY para p_i.

De acordo com a especificação do algoritmo, no máximo $n - k - 1$ processos irão tomar a primeira decisão e permitir imediatamente o uso do recurso, uma vez que k processos estão utilizando os recursos (e adiarão a resposta) e o processo solicitante não contribui para a decisão. Assim, p_i não terá acesso a um recurso e terá que aguardar por uma resposta adiada, que será enviada apenas quando um dos processos que estão utilizando recursos efetuar a liberação. Com isso, garante-se que, no máximo, k processos estarão utilizando os k recursos em cada instante de tempo.

A única exceção que comprometeria a propriedade de *liveness* seria o caso em que o processo solicitante ficasse bloqueado indefinidamente na linha 12 do algoritmo, aguardando pela quantidade mínima de respostas exigida. No entanto, se existe um recurso livre, em algum momento as respostas de autorização serão recebidas pelo processo solicitante, de acordo com as possíveis decisões já discutidas anteriormente. Se algum processo falhar durante a operação, em algum momento o algoritmo de monitoramento irá detectar a falha e atualizar a lista de suspeitos, permitindo que o solicitante não mais aguarde pela sua resposta. Se dois ou mais processos disputam um recurso, a propriedade em questão é garantida pela ordenação do relógio lógico. Logo, todo processo que solicitar um recurso o conseguirá em um tempo finito, de acordo com a ordem de solicitação ou, em caso de empate, com a prioridade do seu identificador.

5. Avaliação Experimental

O algoritmo de k-exclusão mútua proposto foi implementado no *framework* Neko [Urbán et al. 2002]. O Neko é uma ferramenta desenvolvida para a simulação e emulação de algoritmos distribuídos baseado em microprotocolos e troca de mensagens. Cada protocolo é instanciado em um processo (que pode representar um *nodo*) e utiliza uma rede real ou simulada para se comunicar com protocolos em outros processos. No contexto das simulações elaboradas, os algoritmos de exclusão mútua e o algoritmo de monitoramento são protocolos executados em cada processo. Uma camada intermediária foi inserida entre o o algoritmo de exclusão mútua e a rede para transportar as mensagens de REQUEST com base na *spanning tree*.

O algoritmo hierárquico de monitoramento (Hi-ADSD) foi implementado utilizando-se as classes de detecção disponíveis no pacote de tolerância a falhas do Neko. As falhas de processo foram geradas com o mecanismo de *crash* proposto em [Rodrigues e Jansh-Pôrto 2008]. A Figura 4 representa a arquitetura utilizada na construção do ambiente simulado. O algoritmo de k-exclusão mútua (Mutex) envia as mensagens *broadcast* de REQUEST que são interceptadas pela camada responsável pela propagação na árvore (Span). O algoritmo Hi-ADSD envia e recebe mensagens de ARE-YOU-ALIVE e I-AM-ALIVE diretamente na rede. Além disso, toda vez que o estado de um dos processos monitorados é modificado, o Mutex é notificado.

Para o estudo de caso, foram utilizados 16 processos e 5 recursos. A Figura 5 apresenta o hipercubo e a árvore correspondente para o caso em que todos os nodos estão sem-falha. O algoritmo de Raymond e a solução proposta foram configurados para enviar solicitações de recurso periodicamente. Após a obtenção do recurso, o processo aguarda um intervalo de 0,8 para liberá-lo. Em seguida, aguarda outro intervalo de 0,1 e executa uma nova solicitação. A rede do Neko utilizada foi a BasicNetwork, que é simulada e foi configurada com um tempo de transmissão constante de 0,01. O *timeout* para espera

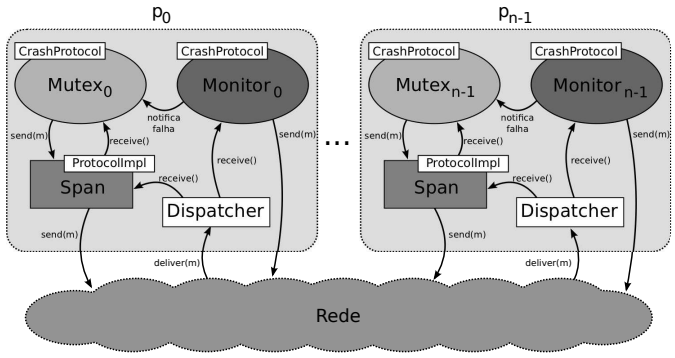
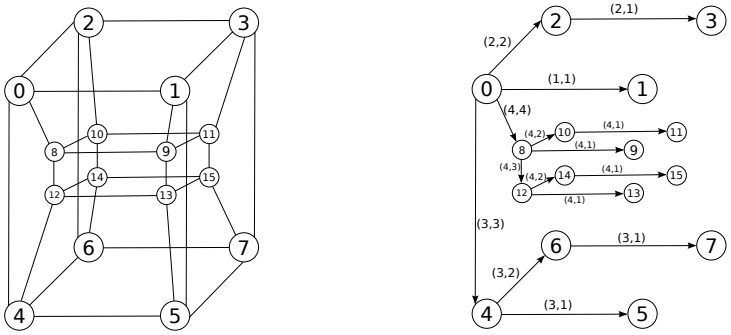


Figura 4. Organização dos módulos de simulação no Neko.



(a) ligações lógicas (b) árvore a partir de n_0

Figura 5. Hipercubo de 4 dim. e a árvore quando todos os nodos estão sem-falha.

dos ACKs foi estabelecido em $2 * \log n$ vezes o tempo de ida e volta, ou seja, 0,08. O Hi-ADSD também é executado periodicamente em intervalos de 1,0 com *timeout* de 0,04.

Falhas *crash* foram injetadas a cada 5,0 intervalos de tempo a partir do tempo 5,0. Em cada intervalo um processo falha, iniciando do processo 16 (p_{15}) até o processo 1 (p_1). Cinco processos (p_0 a p_4) solicitam os recursos periodicamente. As próximas seções apresentam os resultados de número de mensagens e eficiência para o cenário descrito acima, comparando o algoritmo de Raymond com o algoritmo proposto.

5.1. Número de Mensagens

No gráfico da Figura 6 é possível perceber nos intervalos iniciais que o número de mensagens enviadas pelo algoritmo proposto (já com os reconhecimentos do STA) é 50% maior que na solução de Raymond, sem contabilizar as mensagens de monitoramento. Em valores absolutos, o algoritmo proposto enviou 8884 mensagens (3.007 requisições, 2.919 respostas e 2.958 reconhecimentos) ao passo que o algoritmo de Raymond enviou 2.690 mensagens (1.425 req. e 1.265 resp.). Isto representa uma sobrecarga significativa em relação à solução de Raymond, sem considerar as 7.448 mensagens de diagnóstico (5.100 req. e 2.348 resp.). Entretanto, essa diferença deve-se ao fato de que o algoritmo de Raymond ficou bloqueado a partir da quinta falha, deixando de enviar mensagens de requisição após $t = 25,0$. Além disso, pode-se notar que na solução proposta o total de mensagens permanece equilibrado a medida que as falhas são injetadas.

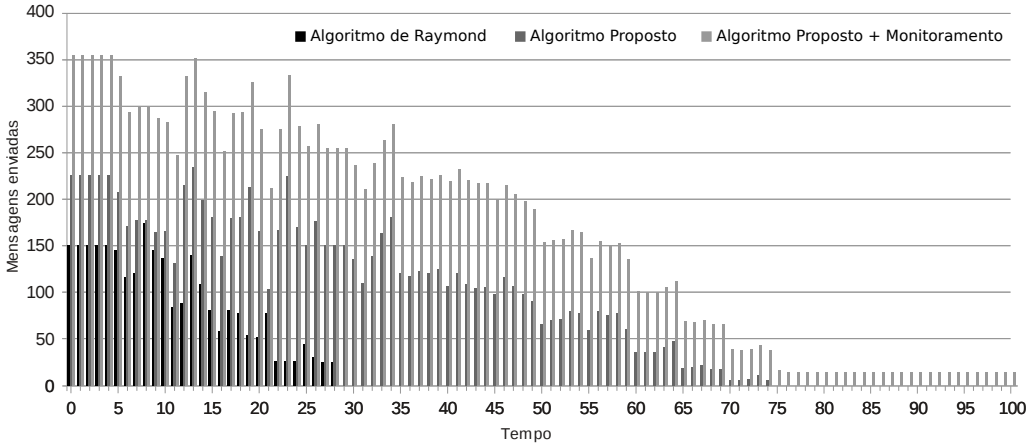


Figura 6. Comparativo de quantidade de mensagens enviadas.

5.2. Eficiência

A Figura 7 mostra a eficiência do algoritmo proposto. Como esperado, após $k - 1 = 4$ falhas o algoritmo de Raymond não consegue mais obter recursos. Isso ocorre após a injeção da quinta falha em $t = 25,0$. Já o algoritmo proposto, com base nas informações fornecidas pelo algoritmo de monitoramento, consegue melhorar significativamente a eficiência até a falha de $n - 1$ processos. Após a falha de $n - k$ processos em $t = 60,0$ ocorre uma degradação constante, que é justificada pela inicio da falha dos cinco processos solicitantes. Após o tempo 75.0 apenas o processo 0 (p_0) continua em execução.

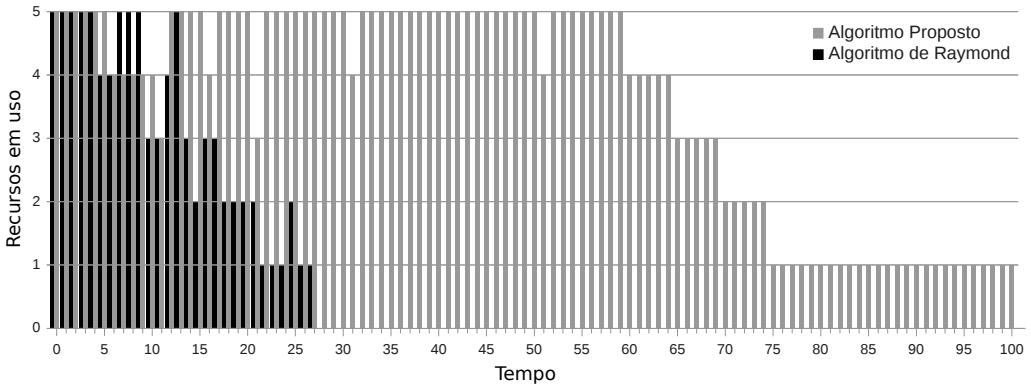


Figura 7. Comparativo de eficiência na obtenção dos recursos.

6. Trabalhos Relacionados

O trabalho de [Naimi 1996] avalia duas abordagens de exclusão mútua em hipercubos. A primeira utiliza pedido de permissão e requer $d2^d$ mensagens, sendo d, a dimensão do hipercubo . A segunda é baseada em passagem de permissão e requer $2d$ mensagens no pior caso. [Raymond 1989a] apresenta um algoritmo baseado em passagem de permissão que utiliza uma *spanning tree*. O número médio de mensagens é $O(\log n)$, mas pode chegar a $2(n - 1)$ no pior caso. Estas propostas, embora otimizadas em termos de quantidade de mensagens, não tratam a falha de processos.

O algoritmo de [Agrawal e El Abbadi 1989] faz uso do conceito de círculo social (*coterie*) introduzido por [Garcia-Molina e Barbara 1985] e de uma organização hierárquica em árvore para resolver a exclusão mútua em $\log n$, sendo n o número de processos. Outra proposta de [Agrawal e El Abbadi 1991] apresenta um algoritmo semelhante e tolerante a falhas. A dificuldade das soluções baseadas neste modelo é a complexidade de construir os círculos.

Mais recentemente, o trabalho de [Bouillaguet et al. 2008] utilizou a abordagem de permissão baseada na proposta de [Raymond 1989b] e que tolera até $n - 1$ processos falhos com o auxilio de um detector de falhas. Posteriormente, [Bouillaguet et al. 2009] propôs uma solução similar que dispensa o uso de detectores de falhas e de mensagens extras para detecção de nodos falhos. As informações de estado dos processos são integradas às mensagens do próprio algoritmo de exclusão mútua, que tolera até $k - 1$ falhas.

7. Conclusão

Este trabalho apresentou uma solução de k-exclusão mútua distribuída com suporte a falhas por $crash$ de até $n - 1$ processos. Um algoritmo distribuído e hierárquico de árvore geradora mínima foi proposto para propagar as mensagens de solicitação de permissão de forma escalável. Os dois algoritmos fazem uso de um mecanismo auxiliar de monitoramento do estado dos processos. O resultado dos experimentos mostrou que, embora as mensagens enviadas pelo monitoramento aumentem significativamente o total de mensagens final, tal custo justifica-se pelo aumento também significativo da eficiência na obtenção de recursos em cenários com falhas.

Como trabalhos futuros, pretende-se avaliar experimentalmente o impacto de falhas no algoritmo da árvore geradora mínima. Além disso, será feita uma comparação da solução proposta com o trabalho de [Bouillaguet et al. 2008].

Referências

Agrawal, D. e El Abbadi, A. (1989). Efficient solution to the distributed mutual exclusion problem. In *Proc. of the 8th ACM Symp. on Princ. Distr. Comp.*, pages 193–200.

Agrawal, D. e El Abbadi, A. (1991). An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Trans. Comput. Syst.*, 9:1–20.

Avresky, D. R. (1999). Embedding and reconfiguration of spanning trees in faulty hypercubes. *IEEE Transactions on Parallel and Distributed Systems*, 10(3):211–222.

Bertsekas, D. P., Ozveren, C., Stamoulis, G. D., Tseng, P. e Tsitsiklist, J. N. (1991). Optimal communication algorithms for hypercubes. *J. P. Distr. Com.*, 11:263–275.

Boehm, H.-J. e Adve, S. V. (2012). You don't know jack about shared variables or memory models. *Commun. ACM*, 55(2):48–54.

Bouillaguet, M., Arantes, L. e Sens, P. (2008). Fault tolerant k-mutual exclusion algorithm using failure detector. In *Int'l Symp. on Parallel and Distr. Comp.*, pages 343–350.

Bouillaguet, M., Arantes, L. e Sens, P. (2009). A timer-free fault tolerant k-mutual exclusion algorithm. In *4th Latin-American Symp. on Dep. Comp.*, pages 41–48.

Bulgannawar, S. e Vaidya, N. (1995). A distributed k-mutual exclusion algorithm. In *Proceedings of the 15th Int'l Conf. on Distr. Comp. Systems*, pages 153–160.

Delporte-Gallet, C., Fauconnier, H., Guerraoui, R. e Kouznetsov, P. (2005). Mutual exclusion in asynchronous systems with failure detectors. *J. P. Distr. Comp.*, 65:492–505.

Duarte Jr., E. P. e Nanya, T. (1998). A hierarchical adaptive distributed system-level diagnosis algorithm. *IEEE Transactions on Computers*, 47(1):34–45.

Gallager, R. G., Humblet, P. A. e Spira, P. M. (1983). A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5:66–77.

Garcia-Molina, H. e Barbara, D. (1985). How to assign votes in a distributed system. *J. ACM*, 32:841–860.

Kruskal Jr., J. B. (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. In *Proc. of the American Mathematical Society*, pages 48–50.

Lamport, L. (1978). Time clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565.

Le Lann, G. (1977). Distributed systems - towards a formal approach. In *IFIP Congress*, pages 155–160.

Naimi, M. (1996). Distributed mutual exclusion on hypercubes. *SIGOPS Oper. Syst. Rev.*, 30:46–51.

Naimi, M., Trehel, M. e Arnold, A. (1996). A log (n) distributed mutual exclusion algorithm based on path reversal. *J. Parallel Distrib. Comput.*, 34:1–13.

Prim, R. C. (1957). Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389–1401.

Raymond, K. (1989a). A distributed algorithm for multiple entries to a critical section. *Inf. Process. Lett.*, 30:189–193.

Raymond, K. (1989b). A tree-based algorithm for distributed mutual exclusion. *ACM Trans. Comput. Syst.*, 7:61–77.

Raynal, M. (1991). A simple taxonomy for distributed mutual exclusion algorithms. *SIGOPS Oper. Syst. Rev.*, 25:47–50.

Raynal, M. e Beeson, D. (1986). *Algorithms for mutual exclusion*. MIT Press, Cambridge, MA, USA.

Ricart, G. e Agrawala, A. K. (1981). An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, 24:9–17.

Rodrigues, L. A. e Jansh-Pôrto, I. E. (2008). Ampliação do framework neko para a simulação de defeitos em algoritmos distribuídos. In *Proc. 7th I2TS*, pages 1–8.

Romano, P. e Rodrigues, L. (2009). An efficient weak mutual exclusion algorithm. In *8th Int'l Symp. on Parallel and Distributed Computing*, pages 205 –212.

Sanders, B. A. (1987). The information structure of distributed mutual exclusion algorithms. *ACM Trans. Comput. Syst.*, 5:284–299.

Suzuki, I. e Kasami, T. (1985). A distributed mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 3:344–349.

Urbán, P., Défago, X. e Schiper, A. (2002). Neko: A single environment to simulate and prototype distributed algorithms. *Journal of Inf. Science and Eng.*, 18(6):981–997.

Memória Compartilhada em Sistemas Bizantinos Dinâmicos*

Eduardo Adilio Pelinson Alchieri[1], Alysson Neves Bessani[2] ,
Joni da Silva Fraga[3] , Fabíola Greve[4]

[1]CIC, Universidade de Brasília, Brasília - Brasil
[2]LaSIGE, Universidade de Lisboa, Lisboa - Portugal
[3]DAS, Universidade Federal de Santa Catarina, Florianópolis - Brasil
[4]DCC, Universidade da Bahia, Bahia - Brasil

Resumo. *Sistemas de quóruns Bizantinos são uma ferramenta útil para a implementação consistente e confiável de sistemas de armazenamento de dados em presença de falhas arbitrárias. Um dos grandes desafios na implementação desses sistemas está na reconfiguração do conjunto de servidores em redes dinâmicas devido à entradas e saídas. Esse trabalho apresenta os principais aspectos do* QUINCUNX*, um conjunto de protocolos tolerantes a faltas Bizantinas capazes de emular registradores em sistemas dinâmicos. O protocolo de reconfiguração é a principal contribuição do artigo e vem a ser o primeiro não baseado no consenso tolerante a faltas Bizantinas. Além disso, seu funcionamento é independente dos protocolos de leitura/escrita, o que permite seu uso com outras implementações de memória compartilhada.*

Abstract. *Byzantine quorum systems are a useful tool to implement consistent and available data storage systems in the presence of arbitrary faults. In this work we consider a dynamic variant of this system and propose a set of Byzantine fault-tolerant protocols, called* QUINCUNX*, that emulates a register in dynamic asynchronous systems. Particularly, we present a reconfiguration protocol that does not require consensus and that is loosely coupled with read/write protocols, making it easy to use with any other static Byzantine fault-tolerant register implementation.*

1. Introdução

Sistemas de quóruns [Gifford 1979] são ferramentas usadas para garantir consistência e disponibilidade de dados, que são armazenados de forma replicada em um conjunto de servidores. Além de serem blocos básicos de construção para protocolos de sincronização (ex.: consenso), o grande atrativo destes sistemas está relacionado com seu poder de escalabilidade e balanceamento de carga, uma vez que as operações não precisam ser executadas por todos os servidores do sistema, mas apenas por um quórum dos mesmos. A consistência de um sistema de quóruns é assegurada pela propriedade de intersecção dos subconjuntos de servidores (quóruns).

Sistemas de quóruns foram inicialmente estudados em ambientes estáticos, onde não é permitida a entrada e saída de servidores durante a execução do sistema [Bazzi and Ding 2004, Malkhi and Reiter 1998a, Malkhi and Reiter 1998b]. Esta abordagem não é adequada para sistemas que permanecerão em execução por um longo tempo, uma vez que, dispondo de um quantidade suficiente de tempo, um adversário pode comprometer um número maior de servidores do que o tolerado e então quebrar as propriedades do sistema. Outra limitação é que estes protocolos não permitem que um administrador, em tempo de execução, adicione máquinas no sistema (para suportar um aumento na carga de processamento) ou troque máquinas antigas. Estes protocolos também não são adequados para sistemas distribuídos modernos, desenvolvidos

*Alysson Bessani é financiado pela FCT através de seu programa multi-anual (LaSIGE) e através dos projetos ReD (PTDC/EIA-EIA/109044/2008) e CMU-Portugal (CMU-PT/0002/2007).

para redes móveis e auto-organizáveis (e.g., MANETs, P2P), onde, pela sua própria natureza, o conjunto de processos que compõem o sistema sofre modificações durante a execução.

A dinamicidade é de fato um dos maiores desafios no projeto dos sistemas de quóruns e, mais especificamente, no projeto de reconfiguração do conjunto de servidores que mantêm a memória compartilhada. Devido à entradas, saídas e falhas dos servidores, será preciso manter a consistência dos dados e garantir a sua disponibilidade. Muitas questões devem ser resolvidas. Há a necessidade de gerenciamento de visões do grupo de servidores que compõem o sistema, mas também da sua quantidade. Se muitos processos deixam o sistema, este pode perder a sua vivacidade. Ademais, o que fazer em caso de falhas sucessivas? E na ocorrência de reconfigurações concorrentes, quando novas reconfigurações se iniciam enquanto outras ainda estão em andamento ? Em qualquer situação, é preciso prover protocolos que mantenham as propriedades de vivacidade e de segurança do sistema, apesar do dinamismo. Esse processo de reconfiguração torna-se ainda mais complexo quando considera-se a possibilidade de componentes maliciosos estarem presentes na computação [Lamport et al. 1982].

Alguns trabalhos foram propostos com o intuito de prover uma memória compartilhada em ambientes dinâmicos e assim implementar um sistema de quóruns dinâmicos [Lynch and Shvartsman 2002, Martin and Alvisi 2004, Rodrigues and Liskov 2004]. Todos esses trabalhos utilizam o protocolo fundamental do consenso como forma de implementar a reconfiguração e assim concordar com o conjunto de servidores que irá compor o sistema. No consenso, todos os processos concordam com uma valor proposto por um dos participantes. Essa abordagem, embora adequada, dado que o serviço de mudança de visões do conjunto de servidores é um protocolo de acordo (ou de consenso), não é a mais eficaz ou a mais indicada. De fato, o consenso torna-se impossível num ambiente assíncrono sujeito a falhas de servidores e, para resolvê-lo, abstrações de sincronia precisam ser incorporadas ao sistema, como os detectores de falhas [Chandra and Toueg 1996]. Ocorre que a manutenção da consistência de uma memória compartilhada com um conjunto estático de servidores pode ser feita sem necessidade de acordo e, portanto, num sistema totalmente assíncrono. Protocolos que implementam sistemas de quóruns em ambientes estáticos [Bazzi and Ding 2004, Malkhi and Reiter 1998a, Malkhi and Reiter 1998b] não precisam de uma primitiva forte de sincronização, como o consenso.

Até bem recentemente, ainda não se sabia se seria possível implementar reconfigurações (ou mudança de visões) sem necessidade de acordo. O trabalho seminal proposto por [Aguilera et al. 2011] responde a essa questão. Ele apresenta o *DynaStore*, um conjunto de algoritmos baseados na abstração *weak snapshot objects*, capaz de implementar uma memória dinâmica tolerante a faltas por parada (*crash*), onde as reconfigurações são executadas sem o auxilio de consenso. Neste sistema, as reconfigurações ocorrem em qualquer momento, gerando um grafo de visões a partir do qual é possível identificar uma sequência de visões onde os clientes executam operações.

Até onde sabemos, o *DynaStore* é o único sistema de memória compartilhada não baseado em consenso. Porém, ele apresenta uma grande desvantagem: o seu protocolo de reconfiguração é fortemente acoplado aos protocolos de leitura e escrita. Um estudo recente [Shraer et al. 2010] confirma que, nesse caso, o desempenho do sistema é fortemente afetado, inclusive, quando comparado com protocolos baseados no consenso. No caso, as leituras e escritas são severamente retardadas em caso de reconfigurações concorrentes com essas operações. Outra característica do *DynaStore*, é que ele somente suporta faltas por parada. Alguns protocolos de reconfiguração foram propostos para o modelo Bizantino [Martin and Alvisi 2004, Rodrigues and Liskov 2004], e de fato, este modelo seria o mais

adequado para redes dinâmicas (MANETs, P2P, etc.), que são altamente vulneráveis a ataques de agentes maliciosos, além de falhas devido à falta de recursos (memória, energia, etc.).

Neste artigo, apresentamos um novo protocolo de reconfiguração que responde às necessidades das redes dinâmicas e possui as seguintes características inovadoras: (i) é o primeiro protocolo de reconfiguração não baseado no consenso capaz de tolerar faltas Bizantinas. Além disso, (ii) o processo de mudança de visões é independente dos protocolos de leitura/escrita. Tal característica contribui para a melhoria de desempenho do sistema, em caso de concorrência das várias operações. A independência permite com que o protocolo de reconfiguração possa ser usado com outras implementações de memória compartilhada.

Neste trabalho, consideramos protocolos para sistemas de quóruns bizantinos do tipo f-disseminação [Malkhi and Reiter 1998b], que toleram faltas maliciosas nos servidores e nos clientes leitores, além de um número ilimitado de faltas por parada de clientes escritores. Os protocolos fazem parte de um sistema chamado QUINCUNX[1], proposto em [Alchieri 2011]. O QUINCUNX implementa um registrador atômico livre de espera (*wait-free*) [Herlihy 1991] e, ao mesmo tempo, permite reconfigurações no conjunto de servidores. Este sistema é composto por dois conjuntos de protocolos: *(1)* protocolos de leitura e escrita; e *(2)* protocolos para atualização de visões. Os protocolos de leitura e escrita são uma variante do PHALANX [Malkhi and Reiter 1998b], que foi a primeira implementação de registrador atômico tolerante a faltas bizantinas. A única diferença é que cada cliente deve verificar se está acessando a configuração mais atual do sistema[2], por isso estes protocolos não são explorados neste trabalho. Já os protocolos para atualização de visões implementam os procedimentos de reconfiguração do conjunto de servidores em tempo de execução e são o foco deste artigo. Para esses protocolos propomos uma abstração chamada *geradores de visões*. Os geradores são classificados de acordo com propriedades de vivacidade (asseguram que um gerador sempre gera alguma visão) e segurança (asseguram que cada gerador gera uma única visão). Neste trabalho, apresentamos o *gerador de visões vivo* que garante que alguma visão sempre será instalada pelo sistema.

O resto deste artigo é organizado da seguinte forma. A Seção 2 apresenta algumas definições preliminares. Os protocolos para reconfiguração do sistema são apresentados na Seção 3. A Seção 4 apresenta algumas discussões sobre os protocolos propostos. As conclusões do trabalho são apresentadas na Seção 5.

2. Definições Preliminares

2.1. Modelo de Sistema

Consideramos um sistema distribuído completamente conectado composto pelo conjunto universo de processos U, que é dividido em três subconjuntos distintos: um conjunto infinito de servidores $\Pi = \{s_1, s_2, ...\}$; um conjunto infinito de clientes leitores $R = \{r_1, r_2, ...\}$; e um conjunto infinito de clientes escritores $W = \{w_1, w_2, ...\}$. Cada processo do sistema (cliente ou servidor) possui um identificador único. Os servidores e os leitores estão sujeitos a faltas Bizantinas [Lamport et al. 1982], enquanto que os escritores apenas podem exibir comportamento de falha por parada. Um processo que apresenta comportamento de falha é dito falho (ou faltoso), de outra forma é dito correto. A chegada dos processos no sistema segue o modelo de chegadas infinita com concorrência desconhecida mas finita [Aguilera 2004].

Consideramos um sistema distribuído assíncrono, onde não existem limites para o tempo

[1]Uma formação militar romana que permite reconfigurações nas linhas de batalha.

[2]Este é o único requisito para qualquer protocolo de leitura e escrita, definido para ambientes estáticos, utilizar os protocolos de atualização de visões do QUINCUNX e transformar-se em um protocolo dinâmico.

de transmissão de mensagens ou processamentos locais nos processos. Além disso, cada servidor do sistema tem acesso a um relógio local usado para iniciar reconfigurações. Estes relógios não são sincronizados e não existe qualquer limite (*bounds*) para seus desvios (*drifts*), sendo portanto apenas contadores. As comunicações entre os processos são realizadas através de canais ponto-a-ponto confiáveis e autenticados. Estes canais são implementáveis em um sistema assíncrono através de SSL/TLS [Dierks and Allen 1999]. Cada servidor possui um par distinto de chaves (chave pública e privada) para usar um sistema de criptografia assimétrica. Todos os escritores também compartilham um par de chaves: as *chaves pública e privada de escrita*. Cada chave privada é conhecida apenas pelo seu próprio dono (ou donos no caso dos escritores), por outro lado todos os processos conhecem todas as chaves públicas. Denotamos uma mensagem m assinada pelo processo i como m_i e consideramos que apenas mensagens corretamente assinadas são processadas pelos processos corretos.

2.2. Dinamicidade e Propriedades do Sistema

Nesta seção definimos as terminologias básicas e as propriedades relevantes para os protocolos do QUINCUNX. Definimos um $update = \{+, -\} \times \Pi$, onde a tupla $\langle +, i\rangle_i$ ou $\langle -, i\rangle_i$ (assinada por i) indica que o servidor i requisitou um *join* (entrada) ou *leave* (saída) do sistema, respectivamente. Dizemos que um *update* foi proposto por i na reconfiguração r caso i execute a operação de *join/leave* antes de r iniciar e nenhuma reconfiguração anterior processou este *update*. Para um conjunto de *updates* u, o conjunto de remoções de u, denotado $u.remove$, é o conjunto $\{i \in \Pi\text{: } \langle -, i\rangle_i \in u\}$. Similarmente, o conjunto de adições de u, denotado $u.join$, é o conjunto $\{i \in \Pi\text{: } \langle +, i\rangle_i \in u\}$. O *membership* de u, denotado $u.members$, é o conjunto $u.join \setminus u.remove$. O *membership* de uma visão v é definido pela união de todos os *updates* computados em v. Um servidor i pertence a v sse $i \in v.members$ (usamos a notação $i \in v$).

Definição 1 (Visão atual no tempo *t*) *Em qualquer tempo t da execução, definimos $V(t)$ como sendo a visão mais atual que algum servidor correto $i \in V(t)$ instalou desde o início de sua execução até t.*

Para qualquer tempo t, os protocolos de reconfiguração do QUINCUNX garantem que pelo menos um quórum de servidores da(s) visão(ões) anterior(es) conhece(m) $V(t)$. Desta maneira , $V(t)$ será a única visão onde clientes podem executar operações de leitura e escrita no tempo t. $V(t)$ permanece *ativa* desde o momento em que foi instalada por i até que uma nova reconfiguração aconteça, de tal maneira que todos os servidores corretos de uma outra visão mais atualizada instalem esta nova visão. Consideramos uma visão inicial $V(0)$ não vazia, que é inicialmente conhecida por todos os processos do sistema no tempo t_0. Também definimos $U(t)$ como sendo o conjunto de *updates pendentes* no tempo t, onde para cada $i \in \Pi$ tal que $\langle +, i\rangle_i \in U(t).join$ temos que $i \notin V(t).join$ e para cada $i \in \Pi$ tal que $\langle -, i\rangle_i \in U(t).remove$ temos que $i \notin V(t).remove$ (i.e., o *update* não foi processado até $V(t)$). Dadas estas definições, as seguintes hipóteses devem ser atendidas para o correto funcionamento do QUINCUNX.

Suposição 1 (Gentle leaves) *Um servidor correto i que requisita um leave no tempo t ($i \in V(t).members \wedge i \in U(t).remove$) permanece no sistema até terminar a instalação de uma visão atualizada na qual i não pertença.*

Esta suposição garante a presença de pelo menos um quórum de servidores corretos em uma dada visão ativa do sistema, garantindo o término das operações e a vivacidade do mesmo.

Suposição 2 (Finite reconfigurations) *O número de updates (join ou leave) requisitados em uma execução é finito.*

Esta suposição permite que as operações de leitura e escrita de clientes, bem como as operações de *join* e *leave* dos servidores, possam terminar apesar das falhas (propriedade de *wait-free*). Para não acessar dados obsoletos, estas operações devem ser executadas na visão mais atual do sistema. Assim, caso um processo muito lento não consiga completar sua requisição antes que a visão do sistema seja atualizada, tal processo deve reiniciar esta requisição utilizando a visão já atualizada. De fato, somente o número de *updates* requisitados concorrentemente com cada operação precisa ser finito. Outros protocolos, como o *DynaStore* [Aguilera et al. 2011], também consideram que requisições de *updates* são finitas. Na verdade, todas as propostas para sistemas de quóruns dinâmicos [Lynch and Shvartsman 2002, Martin and Alvisi 2004, Rodrigues and Liskov 2004] necessitam reiniciar operações devido à reconfigurações concorrentes e, então, devem fazer uso desta suposição para fornecer operações livre de espera (*wait-free*).

Suposição 3 (Fault threshold) *No máximo* $\lfloor \frac{|V(t).members|-1}{3} \rfloor$ *servidores de* $V(t)$ *podem falhar.*

Esta suposição restringe o número máximo de falhas de servidores que podem ocorrer em cada visão do sistema, representando a resiliência ótima para sistemas de quóruns bizantinos de disseminação [Malkhi and Reiter 1998b]: este limite é uma generalização da usual equação $n \geq 3f + 1$, sendo n a cardinalidade do sistema e f o número máximo de falhas. Em qualquer tempo t, somente servidores em $V(t)$ podem enviar respostas para as operações de leitura e escrita executadas por clientes. Quando um servidor i requisita um *join* no sistema, estas operações permanecem desabilitadas em i até a ocorrência de um evento *enable operations*. Após isso, i permanecerá apto para responder estas operações até que i solicite o *leave* do sistema, o que acontece com a ocorrência do evento *disable operations*.

Definição 2 (Propriedades do QUINCUNX) *As seguintes propriedades definem e devem ser garantidas pelo* QUINCUNX*:*

- ***Segurança 1 (Armazenamento)****: Os protocolos de leitura e escrita satisfazem as propriedades de segurança de um registrador de leitura e escrita atômico [Lamport 1986].*
- ***Segurança 2 (Reconfiguração – Join)****: Se um servidor correto* $i \notin V(t)$ *executa a operação de* $join$ *no tempo* t*, então existe um tempo* $t' > t$ *tal que* $i \in V(t')$*.*
- ***Segurança 3 (Reconfiguração – Leave)****: Se um servidor correto* $i \in V(t)$ *executa a operação de* $leave$ *no tempo* t*, então existe um tempo* $t' > t$ *tal que* $i \notin V(t')$*.*
- ***Terminação 1 (Armazenamento)****: Todas operações de leitura e escrita executadas por clientes corretos terminam.*
- ***Terminação 2 (Reconfiguração – Join)****: O evento enable operations terminará por ocorrer em todo servidor correto que executa a operação de* $join$*.*
- ***Terminação 3 (Reconfiguração – Leave)****: O evento disable operations terminará por ocorrer em todo servidor correto que executa a operação de* $leave$*.*

2.3. Visões: Estruturas e Operações

Cada visão v é uma tupla $\langle ov, entries, \mathtt{P} \rangle$, onde: ov é a visão que gerou v, i.e., na reconfiguração de ov a visão v é gerada pelos servidores em ov; $entries$ é um conjunto de $updates$ (Seção 2.2) e define o *membership* de v, como veremos a seguir; e $\mathtt{P}$ é um certificado contendo as provas que garantem tanto a integridade de v como também que v foi instalada no sistema. O campo $entries$ é incremental: para cada visão v, temos que $v.entries = v.ov.entries \cup updates$, onde $updates$ é um conjunto de atualizações válidas usadas para reconfigurar o sistema (Seção 2.2). Como mencionado, $v.entries$ define o *membership* de v: definimos $v.members = \{i \in \Pi : \langle +, i \rangle_i \in v.entries \wedge \langle -, i \rangle_i \notin v.entries\}$ (usamos a notação $i \in v.members$ ou $i \in v$).

Note que um servidor pode entrar e sair do sistema apenas uma única vez, mas podemos relaxar esta condição na prática adicionando números de época em cada requisição de reconfiguração.

***Bootstrapping*.** Na inicialização do sistema, cada servidor ativo $i \in V(0)$ recebe a visão inicial $v_0 = \langle \perp, u_0, \emptyset \rangle$, onde $u_0 = \{\langle +, j \rangle_j : j \in V(0)\}$. $V(0)$ é conhecida por todos os processos do sistema, não sendo necessárias provas para garantir sua integridade.

Limites de falhas e tamanho dos quóruns. Para cada visão v, denotamos $v.f$ como sendo o número de falhas toleradas em v. Seguindo a Suposição 3, $v.f = \lfloor \frac{|v.members|-1}{3} \rfloor$. Nossos protocolos utilizam quóruns de tamanho $v.q = \lceil \frac{|v.members|+v.f+1}{2} \rceil$ [Malkhi and Reiter 1998b].

Comparando e validando visões. Comparamos duas visões v_1 e v_2 através de seus campos $entries$. Usamos a notação $v_1 \subset v_2$ e $v_1 = v_2$ como abreviação de $v_1.entries \subset v_2.entries$ e $v_1.entries = v_2.entries$, respectivamente. Caso $v_1 \subset v_2$, então v_2 é *mais atual* do que v_1. Uma visão é válida caso possua as provas de que foi gerada e instalada no sistema (Seção 3.2.2).

3. Protocolos de Atualização de Visões

Esta seção apresenta os protocolos para reconfiguração do sistema, que são divididos em duas partes: protocolos para geração de uma nova visão e protocolos para sua instalação no sistema.

3.1. Geradores de Visões

Geradores de visões são abstrações utilizadas pelos servidores do sistema na obtenção de novas visões. Para cada visão instalada no sistema v, existe um gerador de visões associado, o qual gerará uma visão atualizada w. Um gerador de visões é acessado através de duas primitivas: (1) $generate_view(v, updates)$, usada pelos servidores de v para iniciar a geração de uma nova visão contendo as atualizações contidas em $updates$; e (2) $new_view(v, w)$, função usada pelo gerador de visões para informar aos processos que a nova visão atualizada w foi gerada.

Definição 3 (GERADORES DE VISÕES) *Os geradores de visões são definidos pelas seguintes propriedades:*

- ***Exatidão**: existem duas variantes desta propriedade:*
  - ***Exatidão forte**: o gerador de visões gera uma única visão w em todos os processos corretos.*
  - ***Exatidão fraca**: o gerador de visões pode gerar diferentes visões em diferentes processos. No entanto, para qualquer duas visões w e w' geradas, temos que $w \subset w' \vee w' \subset w$.*
- ***Terminação**: após inicializado, o gerador de visões acabará por gerar uma nova visão.*
- ***Não trivialidade**: para qualquer visão w, gerada pelo gerador de visões associado a um visão v, temos que $v \subset w$.*

As propriedades de exatidão estão relacionadas com a quantidade de visões geradas (segurança na geração de visões), enquanto que a propriedade de terminação abstrai a possibilidade de um gerador de visões não terminar. A combinação destas propriedades define quatro classes de geradores de visões, apresentadas na Tabela 1. A propriedade de não trivialidade deve ser satisfeita por qualquer gerador, garantindo que as visões geradas são sempre mais atuais.

Geradores das classes $\mathcal{P}$ e $\mathcal{S}$ são considerados seguros, pois os mesmos geram uma única visão (propriedade de exatidão forte). Já os geradores das classes $\mathcal{P}$ e $\mathcal{L}$ são considerados vivos, pois os mesmos sempre geram alguma visão. Os geradores da classe $\mathcal{W}$ são considerados fracos pois podem não terminar (gerar uma visão) ou ainda gerar várias visões. Nesse artigo, apresentamos protocolos para implementação da classe $\mathcal{L}$, com respectiva reconfiguração do QUINCUNX. Em [Alchieri 2011], o leitor encontrará os protocolos que implementam as demais classes (geradores $\mathcal{P}$, $\mathcal{S}$), além dos respectivos protocolos para reconfiguração do sistema.

Tabela 1. Classes de geradores de visões.

Terminação	Exatidão	
	Forte	Fraca
Sim	*Perfeito* $\mathcal{P}$	*Vivo* $\mathcal{L}$
Não	*Seguro* $\mathcal{S}$	*Fraco* $\mathcal{W}$

3.1.1. Gerador de Visões Vivo

O gerador de visões vivo garante que, após ser inicializado, uma nova visão acabará sendo gerada. No entanto, este gerador não garante que a mesma visão será gerada em todos os processos. O Algoritmo 1 apresenta uma implementação desse gerador para sistemas assíncronos.

Algoritmo 1 Gerador de visões vivo (servidor $i \in v$)

variables: {Sets and arrays used in the protocol}
PROP_PR - proofs for the new view entries proposal
NVENTR, NVENTR_PR - expected next view entries and its corresponding proofs

procedure $generate_view(v, updates)$
1: obtain signed tuples $\langle updates, v, i\rangle_*$ from $v.q$ different servers in v and store it on $\texttt{PROP_PR}^v$
2: $\texttt{NVENTR}^v \leftarrow v.entries \cup updates$
3: $\forall j \in v, send\langle$NEW-VIEW, $\langle v, \texttt{NVENTR}^v, \emptyset\rangle_i, \texttt{PROP_PR}^v\rangle$ to j

Upon receipt of $\langle$NEW-VIEW, $\langle v, entries, \emptyset\rangle_j, proof\rangle$ from j
4: **if** $isValidViewProposal(\langle v, entries, \emptyset\rangle_j, proof) \wedge (i, j \in v)$ **then**
5: **wait until** $\texttt{NVENTR}^v \neq \perp$
6: **if** $(\texttt{NVENTR}^v \neq entries) \wedge (entries \not\subset \texttt{NVENTR}^v)$ **then**
7: $\texttt{NVENTR}^v \leftarrow \texttt{NVENTR}^v \cup entries$
8: $\texttt{PROP_PR}^v \leftarrow \texttt{PROP_PR}^v \cup \{proofs\}$
9: $\texttt{NVENTR_PR}^v \leftarrow \emptyset$
10: $\forall k \in v, send\langle$NEW-VIEW, $\langle v, \texttt{NVENTR}^v, \emptyset\rangle_i, \texttt{PROP_PR}^v\rangle$ to k
11: **end if**
12: **if** $\texttt{NVENTR}^v = entries$ **then**
13: $\texttt{NVENTR_PR}^v \leftarrow \texttt{NVENTR_PR}^v \cup \{\langle v, entries, \emptyset\rangle_j\}$
14: **if** $(|\texttt{NVENTR_PR}^v| \geq v.q)$ **then**
15: $new_view(v, \langle v, entries, \texttt{NVENTR_PR}^v\rangle)$
16: **end if**
17: **end if**
18: **end if**

Function $isValidViewProposal(\langle v, entries, \emptyset\rangle_j, proof)$ verifies if:
a) $(\forall e \in entries) \rightarrow (\exists z: \#_{\langle u_z, v, z\rangle_*} proof \geq v.q \wedge e \in u_z) \vee (e \in v.entries)$; and
b) $\#_{\langle u_j, v, j\rangle_*} proof \geq v.q \wedge \forall e \in u_j \rightarrow e \in v.entries$;

O Algoritmo 1 requer 3 passos de comunicação para gerar uma visão. No entanto, pode ser necessário executar mais de uma vez o passo crucial NEW-VIEW. A seguir é descrito como um gerador de visões vivo, associado a uma visão v, gera novas visões para atualização do sistema. Cada servidor correto $i \in v$ certifica sua proposta e envia uma mensagem NEW-VIEW para todos os servidores de v. Os servidores em v atualizam suas propostas caso recebam propostas diferentes (linhas 6-11) e convergem para uma nova visão w assim que os mesmos receberem um quórum de propostas para w (linhas 12-17). Como sempre temos pelo menos um quórum de servidores corretos em v, é garantido que cada servidor $i \in v$ sempre acabará gerando pelo menos uma visão. Vale ressaltar que servidores poderão gerar visões diferentes. No entanto, através das propriedades de intersecção dos quóruns, é garantido que existe uma relação entre qualquer duas visões w e w' geradas por este gerador: $w \subset w'$ ou $w' \subset w$. Assim, estas visões podem ser organizadas em uma sequência, de acordo com o operador "$\subset$".

A Figura 1 apresenta a sequência de visões que pode ser gerada. Como é necessário um quórum de propostas para convergir para uma visão, esta sequência é limitada e conterá no máximo $|v.members| - v.q + 1$ visões: a primeira visão obtida pela computação dos *updates*

de um quórum de servidores ($v.q$) e $|v.members| - v.q$ outras visões obtidas pela adição dos *updates* de cada um dos outros servidores restantes, que devem possuir diferentes propostas para nova visão. As provas de corretude deste algoritmo podem ser encontradas em [Alchieri 2011].

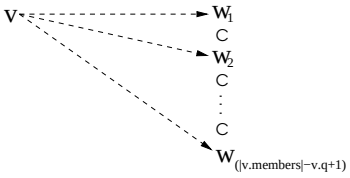


Figura 1. Sequência de visões geradas pelo gerador associado a uma visão v.

3.2. Reconfiguração do Sistema

Esta seção apresenta protocolos para atualizar o conjunto de servidores que implementa o sistema de quóruns, através da utilização do gerador de visões. Estes protocolos não são executados cada vez que um processo solicita um *join*/*leave* do sistema, mas periodicamente quando o tempo de duração de determinada visão se esgotar e existirem requisições de reconfiguração pendentes. Esta abordagem diminui as necessidades de processamento ao executar estas requisições em lotes dentro de uma única atualização de visão. Além disso, esta solução desacopla os protocolos de reconfiguração dos protocolos para acesso (leitura e escrita) ao sistema: as reconfigurações do QUINCUNX podem ser executadas em paralelo com os protocolos de leitura e escrita no registrador, melhorando o desempenho do sistema quando comparado com soluções onde estes protocolos são fortemente acoplados (ex.: [Aguilera et al. 2011]).

3.2.1. Iniciando o Gerador de Visões

Esta seção discute os procedimentos que fazem com que todos os servidores da visão corrente do sistema cv iniciem o gerador de visões associado a esta visão, a fim de reconfigurar o sistema. Este protocolo, apresentado no Algoritmo 2, é utilizado para iniciar qualquer gerador de visões, sendo que apenas os procedimentos após as visões serem geradas são diferentes e dependem do gerador utilizado. Primeiramente, processos devem requisitar sua entrada ou saída do sistema, enquanto cv estiver ativa. Após o tempo de duração de cv se esgotar, o seu gerador de visões associado é iniciado para que novas visões sejam geradas e o sistema reconfigurado.

Requisitando a entrada e/ou saída do sistema. Caso um servidor j desejar entrar no sistema, o mesmo deve encontrar uma visão com algum(s) membro(s) em comum com a visão corrente do sistema cv e executar a operação de $join$ (linas 1-2), onde j envia a tupla assinada $\langle +, j \rangle_j$ indicando que deseja entrar no sistema. Por outro lado, para sair do sistema, j executa a operação de $leave$ (linhas 3-4) e envia a tupla assinada $\langle -, j \rangle_j$. Quando um servidor i receber estas mensagens, i verifica se ambos os servidores (j e i) estão usando a mesma visão mais atual. Caso a visão de j esteja desatualizada, i envia a visão corrente para j, o qual executa novamente sua operação utilizando a visão atualizada (por simplicidade, este processamento não aparece no algoritmo). Do contrário, e caso j ainda não executou a operação de $join$ ou $leave$, i adiciona esta requisição de atualização em seu conjunto de *updates* (`RECV`), para ser processada na próxima reconfiguração do sistema, e envia uma mensagem para j confirmando que a requisição foi recebida (linhas 5-6).

Iniciando o gerador de visões. Cada servidor $i \in cv$ inicia uma reconfiguração para cv através do envio de uma mensagem START-RECONF (linha 7), o que acontece quando o tempo de

duração de cv se esgotar e i possuir *updates* para propor (caso contrário o *timeout* para cv é reiniciado) ou quando i receber mais do que $cv.f$ destas mensagens de outros servidores em cv (linha 9), garantindo que o tempo de duração de cv se esgotou em pelo menos um servidor correto. Assim, servidores maliciosos não são capazes de iniciar uma reconfiguração prematuramente. Cada mensagem START-RECONF contêm a visão corrente do emissor e o conjunto de *updates* proposto para ser aplicado nesta visão. Quando um servidor i recebe estas mensagens, i adiciona os *updates* recebidos em seu conjunto `RECV` (linha 8), a fim de realizar uma proposta com todos os *updates* conhecidos e, principalmente, para garantir que i iniciará a reconfiguração quando o próximo *timeout* para cv se esgotar, pois `RECV` $\neq \emptyset$. Um servidor i espera por $cv.q$ destas mensagens para então iniciar seu gerador de visões (linha 10), propondo todos os *updates* conhecidos até o momento para atualização de cv. Desta forma, todos os servidores corretos de cv iniciarão o gerador de visões associado a esta visão.

Algoritmo 2 Iniciando o gerador de visões (servidor *i*)

variables: {Sets and view generator object used in the protocol}
`RECV` - set of received updates
`VIEW_GEN` - the view generator

procedure $join(v)$
1: $\forall j \in v, send\langle$RECONFIG, $\langle +, i\rangle_i, v\rangle$ to j
2: **wait** for $v.q$ $\langle$REC-CONFIRM$\rangle$ replies from different servers in v

procedure $leave()$
3: $\forall j \in cv, send\langle$RECONFIG, $\langle -, i\rangle_i, cv\rangle$ to j
4: **wait** for $cv.q$ $\langle$REC-CONFIRM$\rangle$ replies from different servers in cv

Upon receipt of $\langle$RECONFIG, $\langle *, j\rangle_j, cv\rangle$ from $j \wedge \langle *, j\rangle_j \notin cv$
5: `RECV` $\leftarrow$ `RECV` $\cup \{\langle *, j\rangle_j\}$
6: $send\langle$REC-CONFIRM$\rangle$ to j

Upon ((timeout for cv and `RECV` $\neq \emptyset$) $\vee$ (start)) $\wedge$ (not started reconfig. for cv)
7: $\forall j \in cv, send\langle$START-RECONF, `RECV`, $cv\rangle$ to j

Upon receipt of $\langle$START-RECONF, $recv, cv\rangle$ from $j \in cv \wedge (\forall u \in recv \rightarrow u \notin cv)$
8: `RECV` $\leftarrow$ `RECV` $\cup\, recv$
9: **if** (received START-RECONF from $cv.f + 1$ servers) **then** start $\leftarrow true$ **end if**

Upon receipt of $\langle$START-RECONF, $recv, cv\rangle$ from $cv.q$ different servers in cv
10: `VIEW_GEN`.$generate_view(cv,$ `RECV`$)$

3.2.2. Reconfiguração com Geradores de Visões Vivos

Esta seção apresenta o protocolo para reconfiguração do sistema através de um gerador de visões vivo $\mathcal{L}$. Apesar da possibilidade de várias visões serem geradas por este gerador, apenas algumas delas são instaladas no sistema, formando uma sequência de visões instaladas. Os clientes apenas poderão executar operações nas visões que formam esta sequência, o que garante as propriedades tanto de terminação quanto de segurança do QUINCUNX.

A principal ideia deste protocolo é fazer com que os servidores organizem as várias visões geradas em sequências. O ponto fundamental para entender o funcionamento deste protocolo é a separação do algoritmo que gera visões atualizadas (encapsulado nos geradores de visões) do algoritmo que organiza estas visões em sequências (apresentado nesta seção). Deste modo, cada visão v estabelece uma sequência de visões para sua atualização e encaminha estes dados para que a visão seguinte na sequência w seja instalada. Após isto, os servidores de w utilizam os dados recebidos de v e definem uma nova sequência de visões para atualização de w, e assim por diante até que existam visões mais atuais para serem instaladas no sistema. Como não utiliza consenso, mais de uma sequência seq_1 e seq_2 de atualização podem ser definidas por uma visão v. No entanto, através da propriedade de intersecção dos quóruns é possível garantir que uma sequência sempre estará contida em outra definida posteriormente, i.e., $seq_1 \subset seq_2$.

Por exemplo, seq_1 pode ser $w_1 \rightarrow w_2$ e então seq_2 é composta por $w_1 \rightarrow w_2 \rightarrow w_3$ (ou $w_0 \rightarrow w_1 \rightarrow w_2 \rightarrow w_3$, ou $w_1 \rightarrow w_2 \rightarrow w_3 \rightarrow w_4$, etc...). Note que todas estas sequências podem ser agrupadas em uma única sequência que engloba todas as visões geradas por determinado gerador, uma vez que as mesmas possuem uma relação que permite esta estruturação. Dada uma sequência seq definida para atualização de uma visão v, apenas a última visão w de seq é efetivamente instalada no sistema (em nosso exemplo: seq_1 instalará w_2 e seq_2 instalará w_3). As visões restantes apenas são utilizadas no processo de instalação de w, pois como é possível que uma sequência anterior definida para atualização de v tenha instalado uma visão $w' \in seq$, o valor do registrador dos servidores em w deve ser atualizado através de uma leitura "realizada em cadeia" nesta sequência (cada visão lê o valor armazenado na visão imediatamente anterior na sequência). Em nosso exemplo, quando da instalação de w_3 (instalada pela seq_2) uma leitura deve ser realizada em w_2, que lê de w_1, que lê de v. Note que w_2 pode ter sido instalada anteriormente através de seq_1, neste caso qualquer valor mais atual escrito em w_2 será enviado para w_3, não sendo perdido durante a atualização da visão do sistema.

As visões que não são instaladas no sistema, chamadas de *visões auxiliares*, não possuem um gerador de visões associado, pois não é necessário atualizá-las (estas visões já se encontram no meio de alguma sequência de visões, i.e., já existe uma visão mais atual para ser instalada no sistema). Desta forma, apenas visões instaladas possuem geradores de visões para geração de novas visões. Neste sentido, um ponto fundamental para o funcionamento correto deste algoritmo é que cada visão efetivamente instalada no sistema pode mudar uma sequência previamente definida para sua atualização, definindo uma nova sequência que "mistura" visões de sequências previamente definidas com visões geradas por seu gerador de visões associado. Também é possível que novas visões sejam criadas através da união de visões previamente definidas com visões geradas pelo seu gerador. Com isto, é possível resolver impasses e organizar todas as visões em uma única sequência. Em nosso exemplo, a visão w_2 poderia iniciar uma reconfiguração antes de receber seq_2 (que faria o sistema atualizar para w_3). Neste caso, o gerador associado à w_2 poderia gerar uma visão w_{2_1} e não seria possível definir se w_2 atualizaria para w_{2_1} (por meio de seu gerador) ou w_3 (por meio de seq_2). Pior ainda, alguns servidores de w_2 poderiam receber primeiro w_{2_1} e tentar atualizar o sistema para esta visão, enquanto que outros servidores poderiam receber primeiro w_3 (seq_2) e tentar instalar esta visão, causando uma divisão de modo que nenhuma das visões pudesse ser instalada. Nossa abordagem resolve este problema fazendo com que w_2 defina uma nova sequência de atualização do sistema, a qual poderá conter tanto a visão w_3 quando a nova visão w_{2_1} ou ainda uma visão representando a união de ambas $w_3 \cup w_{2_1}$, mantendo uma sequência única de visões, como veremos adiante.

Neste protocolo (Algoritmos 3 e 4), para uma visão v ser válida é necessário que: *(1)* uma sequência contendo v foi gerada (Algoritmo 3) – $v.P$ contém um quórum de assinaturas seq_pr^w geradas por servidores de w, para uma sequência seq^w, e $v \in seq^w$ (sequência contendo v gerada pelos servidores da visão w). Então, deve-se também verificar se w é válida seguindo estes mesmos critérios; e *(2)* v foi instalada (Algoritmo 4) – $v.P$ contém um quórum de tuplas $\langle \text{INSTALLED}, w, v \rangle_*$ assinadas por servidores de w.

Este protocolo funciona da seguinte forma. Quando o gerador de visões reporta a geração de uma nova visão w para um servidor $i \in cv$, o mesmo verifica se já fez qualquer proposta para sequência de atualização de cv (linha 1), para então propor uma sequência contendo apenas w (linhas 2-3). Esta visão w será a única visão proposta por i, mas diferentes servidores podem propor diferentes visões (propriedade de exatidão fraca do gerador de visões vivo) e as mesmas devem ser organizadas em uma sequência.

A forma como os servidores convergem para sequências é semelhante com a forma de

como os mesmos convergem para visões no gerador de visões vivo (Seção 3.1.1). Deste modo, quando um servidor $i \in cv$ receber uma proposta para sequência de visões de outro servidor j, o mesmo verifica se todas as visões propostas por j são válidas, i.e., foram geradas pelo gerador de visões (linha 15 do algoritmo 1) ou durante a organização de outras visões válidas (linha 15 do algoritmo 3). Então, dois casos (não mutualmente exclusivos) são possíveis:

Algoritmo 3 Reconfigurando o sistema com geradores vivos (servidor *i*)

variables: {Sets and arrays used in the protocol}
SEQ, SEQ_PR - proposed sequence of views and its corresponding proofs
LCSEQ, LCSEQ_PR - last converged sequence of views known and its corresponding proofs
VIEW_GEN - the view generator (from Algorithm 2)

Upon VIEW_GEN.$new_view(cv, w)$
1: **if** $\texttt{SEQ}^{cv} = \emptyset$ **then** {$w.ov = cv$ (line 15 of Alg. 1}
2: $\texttt{SEQ}^{cv} \leftarrow \{w\}$
3: $\forall j \in cv, send\langle$SEQ-VIEW, $\texttt{SEQ}_i^{cv}, \emptyset\rangle$ to j
4: **end if**

Upon receipt of $\langle$SEQ-VIEW, $seq_j^{cv}, \langle lcseq, proof\rangle\rangle$ from j {i only executes this message if it remains in cv }
5: **if** $isValidSequenceProposal(seq_j^{cv}, \langle lcseq, proof\rangle) \wedge (i, j \in cv)$ **then**
6: **if** $\exists v \in seq_j^{cv}$: $v \notin \texttt{SEQ}^{cv}$ **then**
7: **if** $\exists v, w$: $v \in seq_j^{cv} \wedge w \in \texttt{SEQ}^{cv} \wedge v \not\subset w \wedge w \not\subset v$ **then** {v and w was generated by different reconfig.}
8: $v \leftarrow w$: $(w \in \texttt{LCSEQ}^{cv}) \wedge (\nexists w' \in \texttt{LCSEQ}^{cv}$: $w \subset w')$ {the most updated view in $\texttt{LCSEQ}^{cv}$}
9: $v' \leftarrow w$: $(w \in lcseq) \wedge (\nexists w' \in lcseq$: $w \subset w')$ {the most updated view in $lcseq$}
10: **if** $v \subset v'$ **then** {$lcseq$ is up to date than $\texttt{LCSEQ}^{cv}$}
11: $\langle \texttt{LCSEQ}^{cv}, \texttt{LCSEQ_PR}^{cv}\rangle \leftarrow \langle lcseq, proof\rangle$
12: **end if**
13: $v \leftarrow w$: $(w \in \texttt{SEQ}^{cv}) \wedge (\nexists w' \in \texttt{SEQ}^{cv}$: $w \subset w')$ {the most updated view in $\texttt{SEQ}^{cv}$}
14: $v' \leftarrow w$: $(w \in seq_j^{cv}) \wedge (\nexists w' \in seq_j^{cv}$: $w \subset w')$ {the most updated view in seq_j^{cv}}
15: $\texttt{SEQ}^{cv} \leftarrow \texttt{LCSEQ}^{cv} \cup \{\langle cv, v.entries \cup v'.entries, \langle v, v'\rangle\rangle\}$
16: **else**
17: $\texttt{SEQ}^{cv} \leftarrow \texttt{SEQ}^{cv} \cup seq^{cv}$
18: **end if**
19: $\texttt{SEQ_PR}^{cv} \leftarrow \emptyset$
20: $\forall k \in cv, send\langle$SEQ-VIEW, $\texttt{SEQ}_i^{cv}, \langle \texttt{LCSEQ}^{cv}, \texttt{LCSEQ_PR}^{cv}\rangle\rangle$ to k
21: **end if**
22: **if** $\forall v \in seq_j^{cv} \leftrightarrow v \in \texttt{SEQ}^{cv}$ **then**
23: $\texttt{SEQ_PR}^{cv} \leftarrow \texttt{SEQ_PR}^{cv} \cup seq_j^{cv}$
24: **if** $(|\texttt{SEQ_PR}^{cv}| \geq cv.q)$ **then**
25: $\langle \texttt{LCSEQ}^{cv}, \texttt{LCSEQ_PR}^{cv}\rangle \leftarrow \langle \texttt{SEQ}^{cv}, \texttt{SEQ_PR}^{cv}\rangle$
26: $v \leftarrow w$: $(w \in \texttt{SEQ}^{cv}) \wedge (\nexists w' \in \texttt{SEQ}^{cv}$: $w' \subset w)$ {the next view in the sequence $\texttt{SEQ}^{cv}$}
27: $\forall k \in cv \cup v, send\langle$INSTALL-SEQ, $v, \texttt{SEQ}^{cv}, \texttt{SEQ_PR}^{cv}\rangle$ to k
28: **end if**
29: **end if**
30: **end if**

Upon receipt of $\langle$INSTALL-SEQ, $v, seq^{ov}, seq_pr^{ov}\rangle$ from $ov.f + 1$ different servers in ov and $i \in ov$
31: $\forall k \in ov \cup v, send\langle$INSTALL-SEQ, $v, seq^{ov}, seq_pr^{ov}\rangle$ to k
Upon receipt of $\langle$INSTALL-SEQ, $v, seq^{ov}, seq_pr^{ov}\rangle$ from $ov.q$ different servers in ov and $i \in \{ov \cup v\}$
32: **reconfigure**$(v, seq^{ov}, seq_pr^{ov})$

Function $isValidSequenceProposal(seq_j^{cv}, \langle lcseq, proof\rangle)$ verifies if:
a) $\forall w \in seq_j^{cv}$ we have that $cv \subset w$ and all views in seq_j^{cv} form a sequence according with the $\subset$ operator;
b) $\forall w \in seq_j^{cv}$ we have one of the following:
1 $w.P$ is a set of $\langle w.ov, w.entries, \emptyset\rangle_*$ tuples and: $\#_{\langle w.ov, w.entries, \emptyset\rangle_*} w.P \geq w.ov.q$
2 $w.P$ is a tuple $\langle v, v'\rangle$ and: *(i)* v and v' satisfy the previous condition (case 1) and *(ii)* $((w.entries = v.entries \cup v'.entries) \wedge (v.ov \neq v'.ov))$.
c) $\#_{lcseq_*} proof \geq cv.q$.

(1) i ainda não fez proposta ou a mesma é diferente da proposta de j, onde alguma visão proposta por j ainda não foi proposta por i (linhas 6-21): Então, dois cenários são possíveis:

• As visões já propostas por i e as novas visões recebidas não formam uma sequência (linhas 8-15): isso pode acontecer quando visões geradas por diferentes geradores devem ser organizadas em uma sequência. Então, i define a mais atual sequência de visões já convergida (linhas 8-12) e determina sua proposta de sequência adicionando uma visão que representa a união das duas sequências (a já proposta por i e a recebida de j - linhas 13-15). Isto resolve o

conflito devido à necessidade de organizar sequência de visões geradas por diferentes geradores.

• As visões já propostas por i e as novas visões recebidas formam uma sequência (linha 17): então, a nova sequência a ser proposta por i é formada por todas estas visões conhecidas.

(2) i fez a mesma proposta de j (linhas 22-29): neste caso i coleta a proposta assinada. Quando i obtém a prova de que um quórum fez esta proposta, i converge para uma sequência de visões e envia uma mensagem para que a visão seguinte na sequência v seja inicializada.

Algoritmo 4 Reconfigurando o sistema com geradores vivos (servidor *i*), continuação...

variables: {Sets and arrays used in the protocol}
RECV - set of received updates (from Algorithm 2)
SEQ - proposed sequence of views (from Algorithm 3)

procedure $reconfigure(v, seq^{ov}, seq_pr^{ov})$
1: $v.P \leftarrow seq_pr^{ov}$ {proofs that v is in the sequence}
2: **if** $i \notin v$ **then** {i is leaving the system}
3: *disable operations* {stops accepting R/W client requests}
4: **end if**
5: **if** $cv \subset v$ **then** {i updates its current view to v }
6: $cv \leftarrow v$
7: **end if**
8: RECV $\leftarrow$ RECV $\setminus v.entries$
9: **if** $i \in ov$ **then** {i is member of the previous view in the sequence}
10: **if** $|seq^{ov} \setminus v = \emptyset|$ **then** {v is the last view in the sequence}
11: $\forall j \in v, send\langle$STATE-UPDATE, $data, ts, \langle$INSTALLED, $ov, v\rangle_i$, RECV$\rangle$ to j
12: **else**
13: $\forall j \in v, send\langle$STATE-UPDATE, $data, ts, \perp$, RECV$\rangle$ to j
14: **end if**
15: **end if**
16: **if** $i \in v$ **then** {i is member of the updated view}
17: **wait** for $ov.q$ $\langle$STATE-UPDATE, $*, *, *, *\rangle$ for different servers in ov
18: **if** $(|\{seq^{ov} \setminus v\}| = \emptyset) \wedge cv = v$ **then** {v is the last view in the sequence}
19: $cv.P \leftarrow cv.P \cup$ the $ov.q$ $\langle$INSTALLED, $ov, v\rangle_*$ received {proofs that v was installed}
20: **end if**
21: $\langle data, ts\rangle \leftarrow$ read from the quorum of STATE-UPDATE messages received
22: RECV $\leftarrow$ RECV $\cup$ {all valid updates requests collected from STATE-UPDATE messages} $\setminus$ $cv.entries$
23: **if** $i \notin ov$ **then** {i is entering on the system}
24: *enable operations* {starts to accept R/W client requests}
25: **end if**
26: $\forall j \in ov$: $j \notin v, send\langle$VIEW-INSTALLED, $v\rangle$ to j
27: **if** $(\texttt{SEQ}^v = \emptyset) \wedge (\{seq^{ov} \setminus v\} \neq \emptyset)$ **then**
28: $\texttt{SEQ}^v \leftarrow \{seq^{ov} \setminus v\}$
29: $\forall k \in v, send\langle$SEQ-VIEW, $\texttt{SEQ}_i^v, \emptyset\rangle$ to k
30: **else if** $\texttt{SEQ}^v = \emptyset$ **then** {v is the last view in the sequence seq^{ov}}
31: start timer for cv {if updated on line 6}
32: **end if**
33: **else** {i is leaving the system}
34: **wait** for $v.f + 1$ $\langle$VIEW-INSTALLED, $v\rangle$ from different servers in v
35: discard keys and halt
36: **end if**

Os servidores aguardam por um quórum de INSTALL-SEQ para então "instalar" (lembrando que v apenas será instalada caso não exista uma visão seguinte em seq) a visão seguinte v de uma sequência seq (linha 32). Pela propriedade de intersecção dos quóruns, qualquer sequência posteriormente obtida seq' conterá seq, de modo que qualquer visão instalada através de seq estará contida em seq' e será acessada na atualização do estado da visão instalada através de seq'. Para instalar v, o sistema é reconfigurado da seguinte forma (Algoritmo 4): *(1)* as provas de que v pertence a uma sequência (linha 1) e foi instalada (linhas 10-14,18-20) são coletadas; *(2)* caso exista alguma visão seguinte em seq, uma sequência para atualização de v é proposta (linhas 27-29); e *(3)* um servidor deixa o sistema somente após garantir que a visão mais atual, para a qual enviou o estado, foi instalada (ou teve seu estado atualizado caso seja uma visão auxiliar – mensagens VIEW-INSTALLED). Por exemplo, caso o algoritmo para atualização de cv produza $seq_1 : v_1 \rightarrow v_2$ e depois $seq_2 : v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow v_3$, alguns servido-

res de cv podem receber primeiro seq_1 e outros seq_2 de forma que alguns enviam as mensagens de atualização de estado antes para v_1 e outros para v_0. Caso saíssem do sistema após o primeiro envio desta mensagem, seria possível que nenhuma das visões v_0 ou v_1 recebesse um quórum destas mensagens, de modo que o sistema perderia vivacidade. As provas de corretude deste algoritmo podem ser encontradas em [Alchieri 2011].

Exemplo de funcionamento do protocolo. Imagine que na reconfiguração de uma visão v_0, o gerador de visões associado a esta visão gera as visões atualizadas v_1, v_2 e v_3, sendo que $v_1 \subset v_2 \subset v_3$. Após isso, os servidores em v_0 geram as sequências $seq_1^{v_0} : v_1 \rightarrow v_2$ e $seq_2^{v_0} : v_1 \rightarrow v_2 \rightarrow v_3$. Então, após v_1 ser inicializada (estado atualizado), a mesma pode gerar as sequências $seq_1^{v_1} : v_2$ e $seq_2^{v_1} : v_2 \rightarrow v_3$. Neste exemplo, v_1 é uma visão auxiliar enquanto que v_2 e v_3 poderão ser instaladas. Os servidores de v_2 que receberem $seq_2^{v_1}$ antes de $seq_1^{v_1}$ também não instalam v_2. Por outro lado, caso os servidores em v_2 receberem primeiro $seq_1^{v_1}$, então v_2 é instalada no sistema. Como estamos em um sistema assíncrono, é possível que os servidores de v_2 iniciem o gerador de visões gv para atualização de v_2 e só então recebam $seq_2^{v_1}$ para atualização de v_2 através da instalação de v_3. Neste caso, é possível que concorrentemente gv gere uma nova visão w_1 para atualização de v_2.

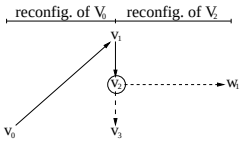
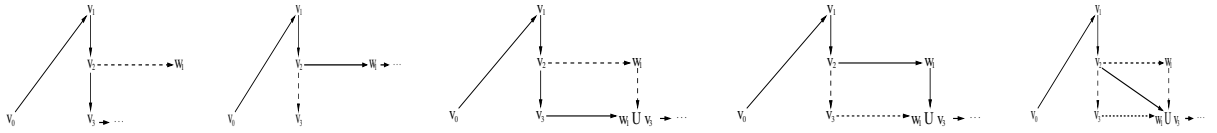


Figura 2. Servidores de v_2 em conflito.

Neste cenário, os servidores de v_2 podem "entrar em conflito" onde alguns recebem primeiro as mensagens para gerar a sequência com w_1 enquanto outros recebem primeiramente as mensagens para gerar a sequência com v_3, como mostra a Figura 2. Neste caso, tanto v_3 quanto w_1 devem ser organizadas em uma sequência a ser instalada a partir de v_2. Para isso, é possível que uma nova visão representando a união destas sequências ($w_1 \cup v_3$) seja gerada. No processamento das mensagens SEQ-VIEW, dependendo de quais mensagens compondo um quorum são entregues nos servidores, é possível que seja instalada apenas uma das duas visões (v_3 ou w_1), ou uma destas visões seguida pela união das mesmas ($w_1 \cup v_3$), ou ainda apenas a união de ambas (Figura 3). Porém, como é necessário um quórum de mensagens para definir uma nova sequência, nunca teremos a instalação de ambas, w_1 e v_3, fazendo com que a sequência de visões instaladas não seja quebrada.



(a) $v_2 \rightarrow v_3$ (b) $v_2 \rightarrow w_1$ (c) $v_2 \rightarrow v_3 \rightarrow w_1 \cup v_3$ (d) $v_2 \rightarrow w_1 \rightarrow w_1 \cup v_3$ (e) $v_2 \rightarrow w_1 \cup v_3$

Figura 3. Possíveis sequências de visões instaladas.

4. Discussão

DynaStore vs. QUINCUNX. No DynaStore, reconfigurações geram um grafo de visões e uma operação de leitura ou escrita precisa percorrer este grafo para encontrar a visão mais atual instalada no sistema, onde é segura a sua execução. Neste processamento, cada aresta deste grafo deve ser acessada para verificar se existe uma visão mais atual instalada no sistema. Para

cada aresta, é necessário acessar um *weak snapshot object*, que é implementado por um conjunto de registradores estáticos (implementados através de sistemas de quóruns). Em contraste, as reconfigurações do QUINCUNX instalam apenas uma sequência de visões, i.e., visões geradas por diferentes geradores são organizadas em uma sequência única.

Usando consenso. Um gerador da classe $\mathcal{P}$ pode ser implementado diretamente através de um protocolo de consenso: quando o gerador é inicializado, as atualizações são propostas em uma instância do consenso de modo que todos os processos decidem pelo mesmo conjunto de atualizações, as quais são aplicadas na visão corrente do sistema, gerando uma nova visão. Caso um gerador desta classe seja utilizado no protocolo de reconfiguração do QUINCUNX, todos geradores geram a mesma visão w no Algoritmo 3 e todos os servidores corretos propõem a mesma sequência $\{w\}$, fazendo o sistema reconfigurar diretamente de cv para w.

5. Conclusões

Este artigo apresentou protocolos de reconfiguração do QUINCUNX, um sistema de quóruns bizantino dinâmico, que implementa um registrador atômico de leitura e escrita. A reconfiguração utiliza um gerador de visões que termina e não se baseia no consenso, além de ser independente em relação aos protocolos de leitura e escrita. Desta maneira, é possível termos concorrência entre estes diversos protocolos, aumentando o desempenho do sistema durante reconfigurações.

Referências

Aguilera, M. (2004). A pleasant stroll through the land of infinitely many creatures. *SIGACT News*, 35(2):36–59.

Aguilera, M. K., Keidar, I., Malkhi, D., and Shraer, A. (2011). Dynamic atomic storage without consensus. *JACM*, 58:7:1–7:32.

Alchieri, E. A. P. (2011). *Protocolos Tolerantes a Faltas Bizantinas para Sistemas Distribuídos Dinâmicos*. Tese de doutorado em engenharia de automação e sistemas, Universidade Federal de Santa Catarina.

Bazzi, R. A. and Ding, Y. (2004). Non-skipping timestamps for Byzantine data storage systems. In *Proc. of 18th Int. Symposium on Distributed Computing, DISC 2004*, volume 3274 of *LNCS*, pages 405–419.

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

Dierks, T. and Allen, C. (1999). The TLS Protocol Version 1.0 (RFC 2246). IETF Request For Comments.

Gifford, D. (1979). Weighted voting for replicated data. In *Proc. of the 7th ACM Symposium on Operating Systems Principles*, pages 150–162.

Herlihy, M. (1991). Wait-free synchronization. *ACM Transactions on Programing Languages and Systems*, 13(1):124–149.

Lamport, L., Shostak, R., and Pease, M. (1982). The Byzantine generals problem. *ACM Transactions on Programing Languages and Systems*, 4(3):382–401.

Lamport, L. (1986). On interprocess communication (part II). *Distributed Computing*, 1(1):203–213.

Lynch, N. and Shvartsman, A. A. (2002). Rambo: A reconfigurable atomic memory service for dynamic networks. In *16th International Symposium on Distributed Computing - DISC*, pages 173–190.

Malkhi, D. and Reiter, M. (1998a). Byzantine quorum systems. *Distributed Computing*, 11(4):203–213.

Malkhi, D. and Reiter, M. (1998b). Secure and scalable replication in Phalanx. In *Proc. of 17th Symposium on Reliable Distributed Systems*, pages 51–60.

Martin, J.-P. and Alvisi, L. (2004). A framework for dynamic Byzantine storage. In *Proceedings of the International Conference on Dependable Systems and Networks*. IEEE Computer Society.

Rodrigues, R. and Liskov, B. (2004). Rosebud: A scalable Byzantine-fault-tolerant storage architecture. MIT-LCS-TR 932, MIT Laboratory for Computer Science.

Shraer, A., Martin, J.-P., Malkhi, D., and Keidar, I. (2010). Data-centric reconfiguration with network-attached disks. In *Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware*.

Caracterização de uma Jornada Máxima em MANETs

Alex Novaes de Santana[1], **Fabíola Greve**[1], **José Garcia Vivas Miranda**[2]

[1]Departamento de Ciência da Computação – Universidade Federal da Bahia (UFBA)
Programa de Pós-Graduação em Mecatrônica

[2]Instituto de Física – Universidade Federal da Bahia (UFBA)
CEP 40210-340 – Salvador – BA – Brazil

alex.santana@gmail.com,fabiola@dcc.ufba.br, vivas@ufba.br

Resumo. *Redes móveis ad-hoc (ou MANETs) exibem uma topologia dinâmica que pode ser modelada através de grafos que evoluem no tempo. Diversos protocolos distribuídos confiáveis, desenvolvidos para tais redes, fazem suposições quanto às características dos grafos necessárias para a satisfação das propriedades do sistema. Um delas diz respeito à difusão de mensagens para a rede, que só poderá ser garantida se, ao longo do tempo, existir um caminho entre cada par de nós do grafo, ou seja uma jornada. Estimar o tempo de uma jornada máxima é essencial para a configuração e qualidade de serviço do sistema. Esse trabalho apresenta um estudo analítico e experimental (através de simulação) para avaliar a conectividade de uma MANET e estimar o tempo de uma jornada máxima, dada as suas características, tais como densidade local e global. Como resultado, apresentamos uma equação probabilística do tempo de jornada máxima no grafo da MANET, em função da sua densidade.*

1. Introdução

As redes móveis ad hoc (ou MANETs) apresentam características e comportamentos que as distinguem das redes tradicionais, a saber: (i) a mobilidade dos dispositivos, bem como a sua dinamicidade (entrada e saída aleatória de dispositivos) ocasiona uma constante mudança topológica na rede, (ii) o grafo de comunicação entre os nós não é completo, o que exige dos nós um esforço colaborativo para transmitir as mensagens; (iii) a estimativa do tempo e condições em que mensagens serão recebidas é dificultada; (iv) ademais, a rede pode particionar em componentes que não se comunicam temporariamente; (v) finalmente, há uma dificuldade na identificação dos parâmetros globais da rede; (vi) o que faz com que cada nó tenha apenas um conhecimento parcial da sua topologia, do conjunto de nós que compõem o sistema e da sua cardinalidade.

Nesse novo cenário, de mobilidade, comunicação parcial e dinamicidade, diversos desafios precisam ser vencidos. Um deles diz respeito à confiança no funcionamento dos sistemas, que permite com que as aplicações móveis distribuídas mantenham a sua disponibilidade e consistência, apesar da ocorrência de falhas (de nós e canais de comunicação), saídas e entradas dos nós. No projeto de sistemas confiáveis, detectar a falha de um processo é um passo importante para garantir a segurança e continuidade correta dos serviços prestados pelo sistema. Em sistemas síncronos a detecção de uma falha é uma tarefa trivial, na medida em que os limites de tempo da rede são conhecidos e garantidos. Porém em redes assíncronas, é preciso projetar uma ferramenta que consiga detectar as falhas mesmo em um ambiente onde os limites de tempo para comunicação são desconhecidos.

Nesse contexto, a abstração de detectores de falhas não-confiáveis é fundamental [Chandra and Toueg 1996]. Informalmente, um detector de falhas é um conjunto de oráculos distribuídos que fornece dicas aos processos sobre quais deles estão falhos, através de uma lista de processos suspeitos. Eles são não-confiáveis porque podem se equivocar, suspeitando da falha de processos corretos ou indicando que processos falhos estão ativos. Os detectores são uma forma modular de tratar as falhas em sistemas assíncronos, isto é, sistemas que não atendam a restrições temporais, dado que separa o tratamento das falhas e os requisitos de sincronia do protocolo distribuído que o utiliza, de forma que este pode lidar apenas com a tarefa a que se propõe.

Em [Greve et al. 2011, Sens et al. 2008] é apresentado um detector de falhas para redes dinâmicas sem fio com mobilidade, tal como as MANETs. O detector proposto, utiliza uma estratégia livre de tempo para a detecção das falhas, baseada numa percepção local dos nós da rede e na troca de mensagens entre os nós vizinhos. Ele faz a suposição de que, apesar da dinamicidade, ao cabo de um tempo, existirá na rede um caminho entre cada par de nós. Essa suposição é importante para garantir que informações globais acerca das suspeitas de falhas sejam difundidas em toda a rede, de forma que suspeitas de falhas possam ser propagadas e suspeitas indevidas possam ser corrigidas. Além dos detectores de falhas, diversos outros protocolos distribuídos confiáveis apresentam tal requisito de conectividade da rede como forma de assegurar a consistência e terminação.

O modelo de grafos estáticos, comumente adotado para redes tradicionais, não representa adequadamente a topologia de redes dinâmicas como as MANETs, já que existe uma variação temporal na topologia da rede, que faz com que as ligações e os caminhos entre os nós sejam construídos ao longo do tempo. Nesse sentido, alguns trabalhos surgiram com o intuito de trazer modelos que melhor se adequem a esta realidade e que possam facilitar a busca e validação de soluções para os algoritmos distribuídos. Com este objetivo [Casteigts et al. 2010a] propuseram um arcabouço para servir como base teórica para o estudo de propriedades fundamentais de algoritmos distribuídos em redes dinâmicas. Este arcabouço permite a caracterização das condições necessárias e/ou suficientes que um algoritmo requer da conectividade da rede, explorando os conceitos de grafos evolutivos [Ferreira 2004, Casteigts et al. 2010b] e renomeação de rótulos [Litovsky and Sopena 1999].

Um outro arcabouço bastante promissor é a abstração de *Time-Varing Graphs* (ou TVG) proposto em [Casteigts et al. 2011a]. Neste, a existência de vértices e arestas é uma função do tempo. No caso de uma rede móvel não-estruturada, a ocorrência das arestas é frequente, mas não segue nenhum padrão definido. A partir dos requisitos apresentados pelos protocolos e aplicações, pode-se identificar classes de TVG que são baseadas nas propriedades temporais estabelecidas pelas entidades. As classes são importantes porque estabelecem condições necessárias (ou mínimas) e resultados de impossibilidade para a computação distribuída. Por exemplo, a classe 3 (conectividade ao longo do tempo) indica um requisito da aplicação que estabelece que o TVG deverá ser conexo ao longo do tempo. Através do uso dessas abordagens [Casteigts et al. 2010a, Casteigts et al. 2011a], pode-se melhor formalizar os algoritmos e protocolos distribuídos e identificar os parâmetros, requisitos mínimos e tipos de redes dinâmicas melhor apropriadas para a sua execução. Por exemplo, em [Casteigts et al. 2010b] foca-se numa análise do problema de difusão (*broadcast*) com detecção de terminação em redes móveis ad-hoc através de uma modelagem

em TVG. Já [Greve et al. 2011] tem como base o modelo do TVG e faz a suposição de que entre qualquer par de nós haverá sempre um *jornada* entre os mesmos. Uma jornada é a formação de um caminho ao longo do tempo entre dois nós.

O objetivo desse artigo é o de avaliar, analítica e empiricamente (através de experimentos de simulação), até que ponto tais suposições quanto à conectividade no tempo associados a uma rede MANET são realistas. Assim, advogamos o uso de um modelo de mobilidade para as MANETs inspirado no modelo de mobilidade humana que se fundamenta no padrão do voo de Levy [Mandelbrot 1983] e apresentamos uma análise do comportamento da MANET a partir de uma modelagem como um TVG. Com base nessa modelagem, balisou-se o grau de conectividade da MANET, através da caracterização do tempo máximo em que uma jornada se forma na rede, variando-se as suas densidades locais e globais. Como resultado, obteve-se uma equação característica do tempo de jornada máxima do TVG em função da sua densidade. A obtenção de tal função característica comprova que numa MANET, apesar da variação da densidade (retirada ou inserção de vértices), ainda existirá uma jornada no tempo entre quaisquer dois nós.

A equação característica obtida pode ser aplicada para aprimorar protocolos distribuídos confiáveis e melhor definir os seus parâmetros relativos à tempo de espera, tempo de difusão para as mensagens, tempo de suspeita, etc. No caso específico de detectores de falhas não-confiáveis, pode-se usar o resultado do nosso estudo para balizar o tempo de emissão de mensagens de *heartbeat* [Friedman and Tcharny 2005] ou tempo de emissão/recepção de mensagens de QUERY-RESPONSE adotadas em [Greve et al. 2011].

O restante do artigo contempla as seguintes seções. A Seção 2 apresenta o modelo de mobilidade e de grafos utilizados para a MANET; a Seção 3 descreve os fundamentos dos detectores de falhas; a Seção 4, descreve o modelo de simulação e cenários; a Seção 5 apresenta os resultados; a Seção 6 os trabalhos correlatos e finalmente a Seção 7 as principais conclusões.

2. Modelo para a MANET

2.1. Modelos de Mobilidade

Um fator importante no estudo, análise e simulação de protocolos e algoritmos para MANETs é o padrão de mobilidade utilizado na construção do modelo da rede. Os modelos de mobilidade são classificados em cinco classes com base em suas características [Khaledi et al. 2009]:

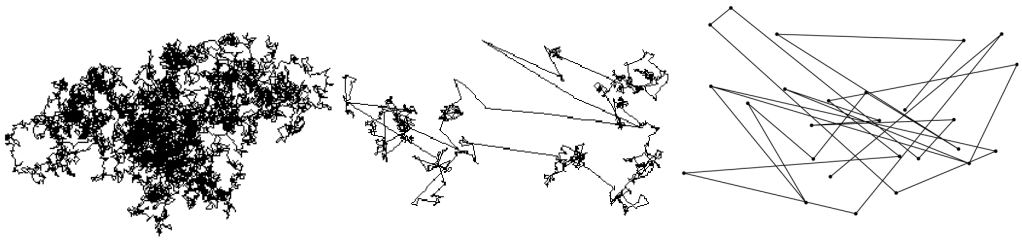
- Classe *Random Models*: Os modelos escolhem aleatoriamente cada novo destino, independente de qualquer interação com a vizinhança, como exemplos de modelos, temos: *Random Walk*, *Random Waypoint*, *Random Direction* e *Levy Flight*.
- Classe *Random Variants Models*: Se diferencia da classe anterior, pois os parâmetros de mobilidades como velocidade e direção são escolhido com base no movimento anterior, exemplos de modelos são: *Gauss-Markov*, *Smooth Random* e *Semi-Markov Smooth Model*.
- Classe *Group Models*: São modelos onde seus elementos se movimentam em grupo, tais como o *Reference Point Group Mobility Model*, *Column*, *Pursue* e *Nomadic Community*.

- Classe dos *Geographic Models*: Modelos que possuem o movimento limitado por caminhos pré-definidos ou obstáculos geográficos, como exemplo temos: *Manhattan*, *Freeway* e *Obstacle*.
- Classe dos *Social Models*: Apresentam um movimento baseado em lugares que concentram elementos por um tempo considerável, chamado ponto de contexto social. Tomando como exemplo uma sociedade real, esses pontos de contexto social seriam centros comerciais, centros de lazer, etc.. Como exemplo de modelos, temos: *CMM*, *Orbit* e *Slaw*.

Se considerarmos uma MANET onde parte dos elementos que constituem a rede são dispositivos associados a um humano (a exemplo de celulares, *smartphones, tablets, notebooks, gps*, etc.), podemos deduzir que o movimento dos dispositivos da rede é determinado pelo modelo de mobilidade humano. Os estudos de [Gonzalez et al. 2008] e [Rhee et al. 2011] demonstraram, através de experimentos reais em diversos cenários, que o modelo de mobilidade humana segue um padrão já observado na natureza, conhecido como voo de Levy [Mandelbrot 1983]. Nesse artigo, advogamos a adoção do modelo de Levy como o mais adequado para modelar as MANETs.

2.2. Modelo de Mobilidade de Levy

O *modelo de mobilidade de Levy* adota uma probabilidade Pr em lei de potência para definir o tamanho das caminhadas, de tal forma que $Pr\,(x) = cx^{-\alpha}$. Isto significa que, quanto maior o tamanho da caminhada x, menor é a probabilidade de sua ocorrência. Desta forma, existirá um número muito maior de pequenas caminhadas, se comparadas ao número de longas caminhas. A título comparativo, enquanto no modelo de *Levy* as caminhadas de menor caminho possuem maior probabilidade, no *Random Waypoint*, a probabilidade é igual para todos os tamanhos de caminhadas. Na Figura 1 podemos observar uma comparação das trajetórias do voo de Levy (1b) com outros dois modelos de mobilidade conhecidos e mais comumente adotados, a saber, o Random Walk (1a) e o Random Waypoint (1c).



(a) Random Walk (b) Voo de Levy (c) Random Waypoint

Figura 1. Trajetórias de Modelos de Mobilidade

2.3. Modelo de Grafos: TVG

A MANET é modelada através de um grafo cuja composição evolui no tempo. Ou seja, a MANET é um *Time-Varing Graph* (ou TVG) [Casteigts et al. 2011b].

Um TVG é representado por uma quíntupla $G = (V, E, T, \rho, \varsigma)$, onde:

- $V = \Pi$ representa o conjunto de vértices;

- $E \subseteq V \times V$ representa as ligações (arestas) entre os vértices;
- T é o tempo de vida do sistema;
- $\rho : E \times T \rightarrow 0, 1$ é a função de presença, que determina se uma aresta está disponível em um determinado instante de tempo;
- $\varsigma : E \times T \rightarrow \mathbb{N}$ é a função de latência, que indica quanto tempo falta, a partir de um tempo $t \in T$, para que uma aresta $e \in E$ esteja disponível.

Dado um TVG, seja $\{(e_1), (e_2)..., (e_k)\}$ um caminho entre dois nós v e w, uma sequencia de pares $J = \{(e_1, t_1), (e_2, t_2)..., (e_k, t_k)\}$ é uma *jornada no tempo* entre v e w, se e somente se $p(e_i, t_i) = 1$ e $t_{i+1} \geq t_i + \varsigma(e_i, t_i)$ para todo $i < k$. Se esta jornada existe, então podemos afirmar que w está ao alcance (no tempo) de v, simplificando $v \rightsquigarrow w$.

3. Detectores de Falhas

Segundo [Chandra et al. 1996], um detector de falhas não confiável é uma espécie de oráculo, capaz de prover informações acerca de processos falhos. Cada nó ou processo possui um detector de falhas local que, quando solicitado, responde com uma lista de processos suspeitos de serem falhos. O detector é considerado não confiável porque lhe é permitido suspeitar de um processo correto ou indicar que um processo faltoso é correto. Porém, o detector também é capaz de corrigir um julgamento errado quando detectado. Um detector de falhas pode ser classificado de acordo com a sua completude e sua acurácia [Chandra et al. 1996]. Entende-se por completude como a capacidade de detectar a falha de um processo. A acurácia está relacionada com a capacidade de não suspeitar indevidamente de processos corretos.

Em [Chen et al. 2002] é apresentado um conjunto de métricas para medir a qualidade de serviço de um detector de falhas. Ela é calculada com base em três métricas: o tempo de detecção de uma falha, o tempo entre a ocorrência de suspeitas erradas e o tempo para a correção de uma suspeita errada.

Na literatura, trabalhos como o de [Friedman and Tcharny 2005, Sridhar 2006] apresentam detectores de falhas para redes móveis baseados em tempo. Em [Friedman and Tcharny 2005], é proposto um detector de falhas que utiliza uma estratégia de *heartbeat*. Nesse tipo de detector, cada processo da rede mantém, enquanto correto, o envio de uma mensagem periódica do tipo "*I am alive*", os processos que ouvem essa mensagem passam a conhecer e não suspeitar do processo emissor. No momento em que um processo qualquer u deixa de ouvir uma mensagem do tipo "*I am alive*" de um processo v, u passa a suspeitar que v falhou.

Em [Greve et al. 2011, Sens et al. 2008], é apresentado um novo tipo de detector de falhas para redes dinâmicas (como as MANETs) não baseado em tempo. Nesse detector, a estratégia utilizada é transmissão/recepção de um par de mensagens do tipo QUERY-RESPONSE. A cada mensagem do tipo QUERY enviada por um processo, este espera por uma quantidade α_i de mensagens do tipo RESPONSE referentes à QUERY efetuada. As mensagens QUERY e RESPONSE carregam consigo duas listas: (i) uma lista de suspeitos e (ii) uma lista de equívocos, contendo suspeitas erradas. A cada troca de mensagens, os processos que respondem à QUERY emitida por v passam a fazer parte do conjunto de processos conhecidos por v, denominado $known_v$. Assim, um processo u em $known_v$, será suspeito no momento em que não responde a uma QUERY executada por

v. A partir desse momento, a cada mensagem trocada, v repassa para os outros processos da rede (através da sua lista de suspeitos) a informação de que u é suspeito. Caso u continue sendo um processo correto e a suspeita de v esteja errada, em um determinada momento, a suspeita chegará até o processo u. Neste momento, o processo u detecta a suspeita indevida e repassa (através da sua lista de equívocos) uma nova informação indicando o erro de julgamento. Com isso, a suspeita errada é corrigida no momento em que u volta a responder a QUERY ou quando v recebe, anexo a uma mensagem *response* de outro processo, a informação de que a suspeita sobre u está errada. Finalmente, a difusão das informações ao longo de toda a rede, sejam estas de suspeita ou de equívocos, só será assegurada caso o TVG (que modela a MANET) esteja conexo no tempo. Ou seja, caso, entre cada par de processos da rede, exista sempre uma jornada no tempo.

Para os detectores de falhas, tanto os baseados em tempo [Friedman and Tcharny 2005, Sridhar 2006], quanto os assíncronos, baseados em emissão/recepção de QUERY-RESPONSE [Greve et al. 2011, Sens et al. 2008], uma ferramenta capaz de (i) identificar se, de fato, a rede dinâmica mantém uma conectividade no tempo, além de (ii) prover uma medição dos limites de comunicação (o tempo de uma jornada máxima) da rede é de grande importância para melhorar a capacidade de detecção de falhas, tornando-a mais rápida e diminuir a quantidade de suspeitas equivocadas, aumentando a sua qualidade de serviço. O modelo e concepção de uma tal ferramenta passa pela identificação de propriedades das MANETs quanto à conectividade, objeto de estudo desse artigo.

4. Implementação do Modelo da MANET

Neste trabalho, o modelo proposto para análise das características de conectividade das MANETs fundamenta-se nos *Time-Varing Graphs* [Casteigts et al. 2011a] e no padrão de mobilidade a partir do voo de Levy [Mandelbrot 1983]. Nesta seção, descrevemos os principais aspectos da implementação deste modelo.

TVG. A rede MANET é um grafo $G = (V, E, T, \rho, \varsigma)$ do tipo TVG, onde o conjunto de vértices $V = \Pi$ representa um conjunto finito de dispositivos móveis, também chamados de nós ou processos. Cada dispositivo possui um identificador único, uma rádio omnidirecional e segue um padrão de mobilidade baseado no voo de Levy [Mandelbrot 1983]. A visão de cada dispositivo está limitada ao alcance do seu rádio de comunicação, ou seja, cada dispositivo conhece a composição da sua vizinhança, mas nenhum dispositivo possui necessariamente o conhecimento global da rede. Desta forma, as arestas em E interligam nós vizinhos. A função de presença $\rho(e, t)$ retorna 1 se, e somente se, no tempo t a distância euclidiana entre os vértices ligados por e é menor que o alcance do rádio de comunicação. Seja R_v o raio de transmissão associado ao nó v, então todos os nós a uma distância máxima R_v de v são considerados seus vizinhos. Seja N_v^t o conjunto de *vizinhos* de v no tempo $t \in T$, então $u \in N_v^t$ *iff* $(v, u) \in E_v^t$, e tal que $\rho((v, u), t) = 1$. A densidade local do nó v é representada pelo *grau* de v no tempo t, dividido por R_v. O grau de v no tempo t é definido por $Deg_v^t = |E_v^t|$.

Os dispositivos da rede estão inseridos em um ambiente em formato de *torus* com tamanho finito W. O formato de *torus* foi escolhido para que não exista, no movimento dos dispositivos, o efeito de tamanho finito [Barabási and Stanley 1995] que modifique

o modelo de voo de Levy. Com relação ao grafo de comunicação formado pelos dispositivos ao longo do tempo, este não necessariamente é conexo ou completamente conectado durante todo o tempo de execução do sistema. Devido à mobilidade, as arestas de comunicação são criadas e desfeitas a todo momento, sem seguir um padrão de ocorrência, o que ocasiona um constante mudança topológica.

Cálculo do Tempo da Jornada. Para calcular a jornada máxima no TVG foi utilizado um protocolo simples de troca de mensagens aliado a um conjunto de lista de identificadores. Cada vértice do TVG possui uma lista que armazena identificadores. Inicialmente, cada lista possui apenas o identificador do vértice ao qual pertence. Em seguida, cada dispositivo executa ciclicamente uma sequencia de três fases: *send*, *receive* e *process*. Durante a fase *send*, o dispositivo envia para todos os seus vizinhos sua lista contendo os identificadores conhecidos. Na fase *receive*, as mensagens enviadas na fase anterior são retiradas do *buffer* de entrada e recebidas por cada dispositivo. Por fim, durante a fase *process*, o dispositivo aplica a operação de união no conjunto de identificadores recebidos e substitui sua lista interna por este conjunto. As listas crescem de forma monotonica à medida que novos nós são descobertos. Caso exista no TVG uma jornada entre cada par de nós da rede, ao cabo de um tempo, por transitividade, cada lista individual irá conter todos os nós da rede. Desta forma, o *tempo da jornada máxima* será o instante de tempo em que todas as listas de identificadores estão completas e contendo todos os nós da rede. Com o objetivo de isolar a influência da densidade na jornada máxima, para a troca das listas foi adotado um sistema de envio de mensagens sem falhas.

A Figura 2 ilustra os passos efetuados por um TVG de 4 nós. Em 2 (a), cada um deles apresenta uma lista contendo apenas os seus respectivos identificadores. Em 2 (b), os nós $v2$ e $v3$ tornam-se vizinhos, mesmo acontece entre $v1$ e $v4$, e promovem a união das suas listas. Finalmente, em 2 (c) mostra-se o instante em que a jornada máxima é descoberta.

Para criar o modelo e TVG e Levy Flight, utilizamos a linguagem c++ para desenvolver o nosso próprio algoritmo.

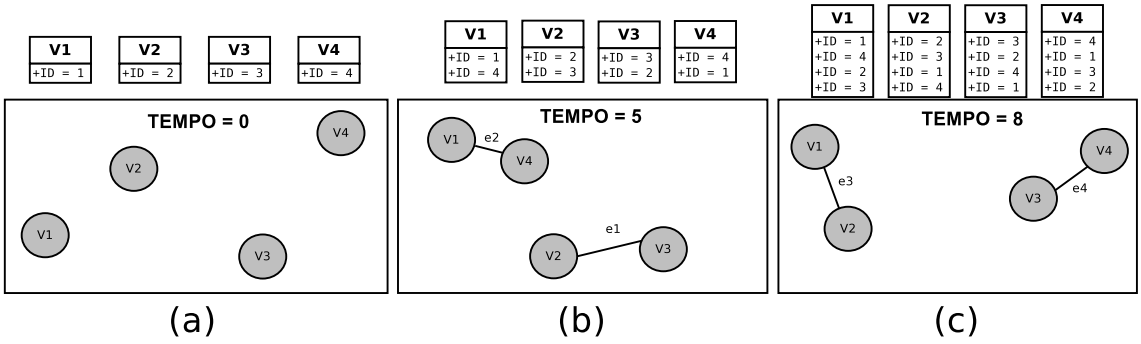


Figura 2. Troca de Listas entre Nós da Rede

4.1. Parâmetros e Cenário de Simulação

Parâmetros. Os parâmetros considerados para implementação do modelo são três: *tamanho do ambiente*: correspondente a área do torus, *densidade global do ambiente*: equivalente ao total de dispositivos dividido pela área do torus e o *parâmetro* α aplicado na

distribuição do voo de Levy. Para esse parâmetro utilizou-se o valor que foi obtido do trabalho de [Gonzalez et al. 2008]. Nesse trabalho, [Gonzalez et al. 2008] monitoraram a trajetória de 100.000 telefones móveis durante um período de seis meses. Ao final do estudo ficou demonstrado que as trajetórias dos usuários, dos telefones monitorados, apresentaram um modelo de voo de levy com $\alpha = 1.75$. Com base neste trabalho, escolhemos o valor de $\alpha = 1.75$ para as simulações do nosso modelo. Os outros dois parâmetros variaram de acordo com a Tabela 1.

Tamanho do ambiente (m^2)		6.25x10^4	2.5x10^5	5.625x10^5	1.0x10^6	
Densidade global	0.0002	0.0003	0.0004	0.0005	0.0006	0.0007
	0.0008	0.0009	0.0010	0.0011	0.0012	

Tabela 1. Parâmetros da Rede

Métricas. Em cada execução foi registrado o *tempo máximo da formação da jornada na rede*, ou seja, o tempo em que todos os dispositivos completaram suas listas de identificadores. Além disso, o número de conexões de cada dispositivo em cada iteração foi registrado para obtermos a *densidade local* de cada dispositivo v no tempo t. i.e., Deg_v^t. Com o objetivo de avaliar a influência de cada parâmetro no cálculo da jornada, em cada combinação dos parâmetros iniciais, o modelo foi executado 300 vezes.

5. Resultados e Avaliação da Conectividade da MANET

Densidade X Jornada Máxima. Os parâmetros iniciais de tamanho do ambiente e densidade foram combinados com o objetivo de comparar a influência da densidade no tempo de jornada máximo em ambientes de tamanhos variados. Para cada combinação foram obtidas trezentas amostras da jornada máxima no TVG. De posse dessas amostras, o próximo passo foi verificar o comportamento característico da jornada máxima com relação à densidade. O gráfico da Figura 3 apresenta as jornadas máximas médias para cada condição inicial do modelo. As barras verticais em cada ponto representam o desvio padrão amostral da simulação de 300 amostras. Analisando os dados em um gráfico semi-log, os dados apresentam o formato de uma reta. Esse tipo de comportamento é ajustável somente com funções exponenciais. Neste caso especificamente, os dados se ajustam a uma função exponencial decrescente, esse comportamento é comprovado através de um ajuste não linear, veja a Figura 3. O ajuste de cada conjunto de dados é representado por uma linha contínua na figura 3. A função 1 foi utilizada no método de ajuste, o relatório com os valores do ajuste pode ser observado na Tabela 2. Na função 1, $y0$ representa o *offset*, A representa a sua amplitude e t representa a constante de decaimento.

Com base na função característica para o tempo de jornada máximo (Função 1), ao se aumentar o tamanho W temos algo equivalente a reduzir a densidade para zero. Neste caso, o limite da Função 1 com a densidade x tendendo a zero é igual a soma de duas constante $y0$ e A. Ou seja, mesmo num ambiente com uma esparsa quantidade de dispositivos, ainda assim, existirá uma jornada de tamanho máximo igual a $y0 + A$.

Consideremos agora o caso contrário, onde a densidade do ambiente tende ao infinito. Neste caso a Função 1 tende para um valor igual a $y0$. Ou seja, mesmo no caso

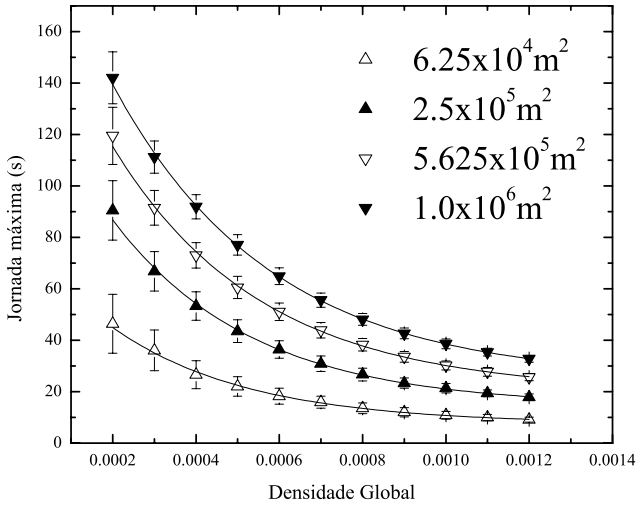


Figura 3. Densidade X Jornada Máxima

W	$y0$	A	t
$6.25x10^4m^2$	7.5074 ± 0.24837	69.0738 ± 2.98027	$32.6698x10^{-5} \pm 1.22$
$2.5x10^5m^2$	14.40617 ± 0.4075	131.5002 ± 3.9395	$335.013x10^{-6} \pm 9.72683$
$5.625x10^5m^2$	20.03765 ± 0.56484	166.84008 ± 4.0372	$357.035x10^{-6} \pm 9.55493$
$1.0x10^6m^2$	24.54946 ± 0.43951	194.43159 ± 2.564	$380.077x10^{-6} \pm 6.09464$

Tabela 2. Relatório de Ajuste

de um ambiente extremamente denso, ainda existirá uma jornada entre dois vértices do TVG.

Esse resultado é importante para garantir a existência da propriedade de conectividade no tempo em redes MANETs, considerando-se um modelo de mobilidade humano. Isso é essencial para garantir o funcionamento correto de alguns dos detectores de falhas citados neste trabalho. Além disso, caracterizar o tempo da jornada máxima é importante para ajudar na escolha da estratégia de detecção durante o desenvolvimento detectores. Uma estratégia adequada diminui o tempo de detecção, recorrência/recuperação de falsas detecções, e finalmente aumenta a qualidade de serviço do detector.

$$y = f(x) = y0 + Ae^{\frac{-x}{t}} \tag{1}$$

$$\lim_{x \to 0} f(x) = y0 + A \tag{2}$$

$$\lim_{x \to \infty} f(x) = y0 \tag{3}$$

Densidade Global X Densidade Local. A Função 1 descreve o comportamento da jornada máxima com base na densidade global do ambiente. Porém, como os vértices da rede possuem apenas uma visão local do sistema, estes não possuem o conhecimento da densidade global. Sem o parâmetro da Função 1, é preciso obter uma relação entre a densidade global e a densidade local. Como densidade local adotamos a quantidade de dispositivos dentro do rádio de comunicação de um dispositivo dividido pela área de alcance do rádio.

Observando o Gráfico 4, que apresenta a média dos graus ao longo do tempo de cada vértices do TVG, podemos verificar que existe uma relação linear entre a densidade global do ambiente e a densidade local do dispositivo. Com isto podemos obter uma densidade global x a partir de uma densidade local x', conforme a Função 4.

Substituindo na Função 1, o valor da densidade global x pela Função 4, obtemos uma relação entre a densidade local e a jornada máxima do TVG.

$$x = g(x') = ax' + b \tag{4}$$

$$f(g(x')) = y0 + Ae^{\frac{-x}{t}} \tag{5}$$

$$f(x') = y0 + Ae^{\frac{-(ax'+b)}{t}} \tag{6}$$

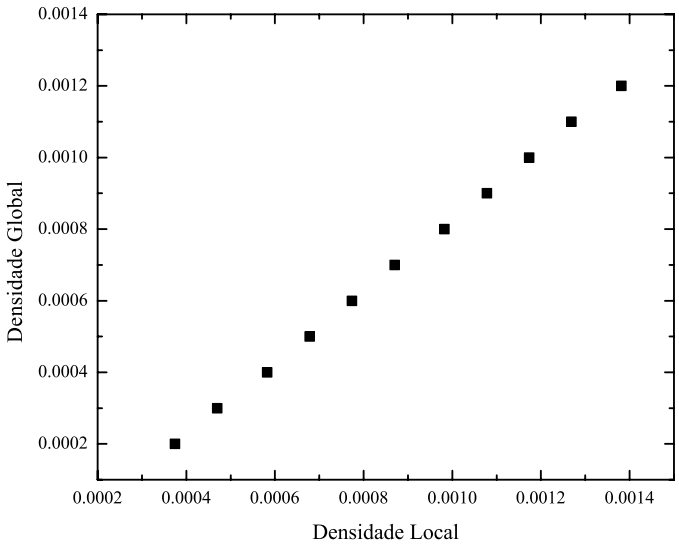


Figura 4. Densidade Global X Densidade Local

6. Trabalhos Relacionados

Alguns outros trabalhos [Ferreira 2004, Bui Xuan et al. 2002] propõem a utilização de grafos evolutivos para modelar as redes dinâmicas. Esta abordagem se diferencia um

pouco da TVG [Casteigts et al. 2011a, Casteigts et al. 2011b], dado que nesta os grafos evoluem com o tempo a partir de uma sequencia de grafos estáticos. Vale destacar que o trabalho de [Ferreira 2004] também apresenta um conjunto de conceitos e métricas sobre jornadas em rede dinâmicas. Com base nessas métricas, [Bui Xuan et al. 2002] apresentam o cálculo de uma jornada mais curta e mais rápida numa rede dinâmica, como por exemplo, MANETs e sistemas de satélites em órbita terrestre baixa.

[Bettstetter 2002] aborda o problema de conectividade em redes sem fio. O autor investiga a distribuição espacial dos dispositivos da rede e o seu efeito na conectividade. O trabalho estima a quantidade necessária de dispositivos para cobrir uma área com o objetivo de obter uma rede k-conexa. Diferentemente do modelo adotado pelo nosso trabalho, baseado na mobilidade humana, os modelos de mobilidade analisados pelo autor foram os *Random Waypoint Model* e *Random Direction Model*.

[Xiang et al. 2010] também apresentam um trabalho analítico de investigação sobre a probabilidade de uma MANET ser k-conexa. Essa característica é importante em protocolos de tolerância a falhas, pois uma rede k-conexa suporta até $k - 1$ falhas de nós sem que ocorra um particionamento da rede. Em [Xiang et al. 2010], foi utilizado o modelo de mobilidade *Random Way Point*.

[Desai and Manjunath 2002] apresentam um estudo sobre o efeito da quantidade de nós e do tamanho do ambiente na probabilidade da rede ser conexa. O estudo concentrou-se em redes ad-hoc de tamanho finito unidimensionais e foi estendido para redes bi-dimensionais.

[Bettstetter 2004] apresenta um estudo analítico sobre limites probabilístico da conectividade em MANETs. O modelo de rede ad-hoc estudada é formada por um conjunto de nós espalhados em uma área bi-dimensional de tamanho finito. Os nós da rede apresentam um modelo de mobilidade do tipo *Random Way Point*. Apresenta-se resultados sob três pontos de vista: (i) visão de um único nó, com o números de vizinhos de um nó; (ii) visão de dois nós, com a probabilidade de existir um caminho entre dois nós; e (iii) uma visão geral da rede, com a probabilidade de existir um caminho entre quaisquer dois nós, ou seja, a probabilidade da rede ser conexa.

[Clementi et al. 2010, Foh et al. 2005] apresentam um estudo sobre o tempo de alcance de um *flooding* em redes ad-hoc. Utilizando o modelo de mobilidade *Manhattan*, o trabalho de [Clementi et al. 2010] apresenta um estudo analítico sobre a propagação de dados numa MANET. Como resultado o trabalho demonstra que o tempo de propagação no "subúrbio", região mais esparsa, é tão rápido quanto na "zona central", região mais densa. Ou seja, no modelo proposto por [Clementi et al. 2010], o tempo de disseminação da informação não sofre alterações significativas em densidades diferentes.

[Foh et al. 2005] apresentam uma caracterização da conectividade em redes ad-hoc. Esse estudo utiliza um modelo de mobilidade do tipo *Random Way Point* sobre um ambiente unidimensional. Diferentemente, nosso trabalho faz um estudo sobre uma área bi-dimensional com um modelo de mobilidade de *Levy Flight* (observado no comportamento humano). [Foh et al. 2005] utilizou um ambiente unidimensional com *Random Way Point*, pois o foco do seu estudo são redes ad-hoc que representam ônibus urbanos andando sobre uma via exclusiva de ônibus com paradas em estações.

[Clementi et al. 2011] apresentam um estudo analítico sobre o limite máximo de

tempo para o espalhamento de informações em redes ad-hoc. Os autores representaram a rede através de um modelo de grafos evolutivos, onde a existência de uma aresta no tempo é determinada por uma cadeia de Markov. Esse tipo de modelo engloba modelos de mobilidade como o *Random Way Point* em uma esfera ou torus. Ele se diferencia do modelo proposto neste trabalho, por não adotar o Levy Flight como modelo de mobilidade.

[Srivastava et al. 2008] apresentam, a partir de dados reais, uma caracterização da topologia e do roteamento numa MANET. Os autores utilizaram dados reais obtidos durante o evento *Mobile Ad-hoc Network Interoperability and Cooperation* (MANIAC), que reúne um conjunto de universidades para o estudo da interoperabilidade e cooperação. Por se tratar de um experimento real, o número de nós em estudo foi relativamente pequeno, cerca de dezesseis nós. Como resultados, este trabalho demonstrou que as redes apresentam um alto grau de conexão e aglomeração, o que resulta em um diâmetro pequeno. Com relação ao roteamento, os dados indicaram uma grande quantidade de *churn* (entrada e saída aleatória de nós), com as rotas sofrendo modificações constantemente.

Considerações. O modelo de mobilidade utilizado pode interferir diretamente na dinâmica de criação e rompimento de novas arestas no grafo. Em comparação com o estudo proposto neste presente trabalho, os trabalhos relacionados abordam modelos de mobilidade diferentes do *Levy Flight*. Além disso, tomando-se por foco a conectividade, os trabalhos de [Bettstetter 2002, Xiang et al. 2010, Desai and Manjunath 2002] procuram descobrir ou definir características da rede necessárias para garantir uma k-conectividade. Por outro lado, nosso trabalho estuda a conectividade com o objetivo de investigar se a rede formada será conexa ao longo do tempo, sem que condições para garantir tal k-conectividade precisem ser impostas. Esses fatores demonstram que o presente trabalho apresenta uma linha de investigação com vários trabalhos relacionados, porém com uma abordagem inovadora, que faz uso de um modelo de mobilidade diferente do empregado usualmente nos trabalhos de investigação sobre redes móveis.

7. Conclusão

Este trabalho estudou o comportamento de uma MANET representada através de um *Time-Varing Graphs* (TVG), considerando-se um padrão de mobilidade baseado no voo de Levy. Acreditamos que, embora não usual, tal modelo de mobilidade seja mais adequado para uma MANET. O seu principal objetivo foi o de avaliar o grau de conectividade da MANET, através do cálculo de tempo das jornadas formadas entre os diversos dispositivos. Como resultados das simulações, obteve-se uma equação característica do tempo de jornada máxima do TVG em função da sua densidade. A obtenção de tal função característica comprova que numa MANET, apesar da variação da densidade (retirada ou inserção de vértices), ainda existirá uma jornada no tempo entre quaisquer dois vértices do TVG. Adicionalmente, a equação característica poderá ser usada como parâmetro para aprimorar protocolos distribuídos confiáveis, e mais particularmente, detectores de falhas não-confiáveis. Nesse caso específico, a equação poderá ser usada para balizar o tempo de emissão de mensagens de *heartbeat* ou tempo de emissão/recepção de mensagens de QUERY-RESPONSE, comumente adotados pelos protocolos de detecção. Como trabalhos futuros, pretendemos usar as medidas e equação obtidas para propor uma implementação eficiente do protocolo de detecção de falhas em [Greve et al. 2011].

Referências

Barabási, A. and Stanley, H. (1995). *Fractal concepts in surface growth*. Cambridge Univ Pr.

Bettstetter, C. (2002). On the minimum node degree and connectivity of a wireless multihop network. In *Proceedings of the 3rd ACM international symposium on Mobile ad hoc networking & computing*, pages 80–91. ACM.

Bettstetter, C. (2004). On the connectivity of ad hoc networks. *The Computer Journal*, 47(4):432–447.

Bui Xuan, B., Ferreira, A., and Jarry, A. (2002). Computing shortest, fastest, and foremost journeys in dynamic networks. Technical Report RR-4589, INRIA.

Casteigts, A., Chaumette, S., and Ferreira, A. (2010a). Characterizing topological assumptions of distributed algorithms in dynamic networks. In *Structural Information and Communication Complexity Conf.*, pages 126–140.

Casteigts, A., Flocchini, P., Mans, B., and Santoro, N. (2010b). Deterministic computations in time-varying graphs: Broadcasting under unstructured mobility.

Casteigts, A., Flocchini, P., Quattrociocchi, W., and Santoro, N. (2011a). Time-varying graphs and dynamic networks. Technical report, University of Ottawa.

Casteigts, A., Flocchini, P., Quattrociocchi, W., and Santoro, N. (2011b). Time-varying graphs and dynamic networks. *Ad-hoc, Mobile, and Wireless Networks*, pages 346–359.

Chandra, T., Hadzilacos, V., and Toueg, S. (1996). The weakest failure detector for solving consensus. *Journal of the ACM (JACM)*, 43(4):685–722.

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

Chen, W., Toueg, S., and Aguilera, M. (2002). On the quality of service of failure detectors. *Computers, IEEE Transactions on*, 51(5):561–580.

Clementi, A., Monti, A., Pasquale, F., and Silvestri, R. (2011). Information spreading in stationary markovian evolving graphs. *Parallel and Distributed Systems, IEEE Transactions on*, 22(9):1425–1432.

Clementi, A., Monti, A., and Silvestri, R. (2010). Fast flooding over manhattan. *Arxiv preprint arXiv:1002.3757*.

Desai, M. and Manjunath, D. (2002). On the connectivity in finite ad hoc networks. *Communications Letters, IEEE*, 6(10):437–439.

Ferreira, A. (2004). Building a reference combinatorial model for manets. *Network, IEEE*, 18(5):24–29.

Foh, C., Liu, G., Lee, B., Seet, B., Wong, K., and Fu, C. (2005). Network connectivity of one-dimensional manets with random waypoint movement. *Communications Letters, IEEE*, 9(1):31–33.

Friedman, R. and Tcharny, G. (2005). Evaluating failure detection in mobile ad-hoc networks. *Int. Journal of Wireless and Mobile Computing*, 1(8).

Gonzalez, M., Hidalgo, C., and Barabási, A. (2008). Understanding individual human mobility patterns. *Nature*, 453(7196):779–782.

Greve, F., Sens, P., Arantes, L., and V.Simon (2011). A failure detector for wireless networks with unknown membership. In *Euro-Par Conference, LNCS 6853*, pages 27–38.

Khaledi, M., Hemmatyar, A., Rabiee, H., Mousavi, S., and Khaledi, M. (2009). Mobility analyzer: a framework for analysis and recognition of mobility traces in mobile ad-hoc networks. In *New Technologies, Mobility and Security (NTMS), 2009 3rd International Conference on*, pages 1–5. IEEE.

Litovsky, I. and Sopena, E. (1999). Graph relabelling systems and distributed algorithms. In *Handbook of Graph Grammars and Computing by Graph Transformation. Concurrency, Parallelism, and Distribution*, volume 3, pages 1–56.

Mandelbrot, B. (1983). *The fractal geometry of nature*. Wh Freeman.

Rhee, I., Shin, M., Hong, S., Lee, K., Kim, S., and Chong, S. (2011). On the levy-walk nature of human mobility. *Networking, IEEE/ACM Transactions on*, 19(3):630–643.

Sens, P., Greve, F., Arantes, L., Bouillaguet, M., and Simon, V. (2008). Um Detector de Falhas Assíncrono para Redes Móveis e Auto-Organizáveis. In *Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, Rio de Janeiro, RJ, Brazil.

Sridhar, N. (2006). Decentralized local failure detection in dynamic distributed systems.

Srivastava, V., Hilal, A., Thompson, M., Chattha, J., MacKenzie, A., and DaSilva, L. (2008). Characterizing mobile ad hoc networks: the maniac challenge experiment. In *Proceedings of the third ACM international workshop on Wireless network testbeds, experimental evaluation and characterization*, pages 65–72. ACM.

Xiang, H., Liu, J., and Kuang, J. (2010). Minimum node degree and connectivity of two-dimensional manets under random waypoint mobility model. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 2800–2805. IEEE.

XIII Workshop de Testes e Tolerância a Falhas

♦

Sessão Técnica 3

Testes, Verificação e Especificação Formal

Uma Abordagem para Geração de Casos de Teste Estrutural Baseada em Modelos

Leandro T. Costa, Flávio M. Oliveira, Elder M. Rodrigues, Maicon B. da Silveira, Avelino F. Zorzo

[1]Programa de Pós-Graduação em Ciência da Computação
Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Caixa Postal 1.429 – 90.619-900 – Porto Alegre – RS – Brasil

leandro.teodoro@acad.pucrs.br, bernardino@acm.org,

{elder.rodrigues, flavio.oliveira, avelino.zorzo}@pucrs.br

Abstract. *Structural testing, or white-box testing, is a technique to generate test cases based on analysis of an application source code. Currently, there are a lot of tools that perform this type of test, e.g. JaBUTi and Poke-Tool. However, despite the benefits of these tools, some tasks still have to be performed manually, e.g. the description of the test cases. This makes the test process inefficient and prone to faults. Thus, this paper describes a technology-independent test case format, i.e., an abstract structure for white-box testing. We describe also how this abstract structure is inserted into a process that uses it to, automatically, generate and execute concrete test cases in a specific structural testing tool*[1][2].

Resumo. *O teste estrutural, ou teste caixa-branca, é uma técnica para gerar casos de teste a partir da análise do código fonte. Atualmente, existem muitas ferramentas baseadas neste tipo de teste, e.g. JaBUTi e Poke-Tool. No entanto, apesar dos benefícios destas ferramentas, é necessário executar algumas tarefas manualmente, e.g. a elaboração de casos de teste. Isto torna o processo de teste ineficiente e suscetível a falhas. Desta forma, este artigo apresenta um formato para casos de testes independente de tecnologia, i.e., uma estrutura abstrata para teste estrutural. O artigo descreve, também, como esta estrutura abstrata é inserida em um processo que a usa para, automaticamente, gerar e executar casos de teste concretos em uma tecnologia de teste estrutural específica.*

1. Introdução

A evolução e incremento da complexidade dos sistemas computacionais existentes têm tornado o processo de teste destes sistemas uma atividade tão ou mais complexa que o processo de desenvolvimento em si. Para contornar os problemas decorrentes do aumento da complexidade dos sistemas, e aumentar a eficiência no processo de geração de testes das aplicações, diversas ferramentas foram desenvolvidas com o objetivo de automatizar os processos de avaliação e verificação de *software*. Atualmente, existem diversas ferramentas que avaliam a funcionalidade, desempenho, disponibilidade e escalabilidade das

[1]A ordem dos autores é meramente alfabética.

[2]Study developed by the Research Group of the PDTI 001/2012, financed by Dell Computers of Brazil Ltd. with resources of Law 8.248/91.

aplicações. Dentre elas citam-se Apache JMeter [Halili 2008], IBM Rational Performance Tester (RPT) e JaBUTi (*Java Bytecode Understanding and Testing*) [Eler et al. 2009].

No entanto, apesar dos benefícios advindos do uso de ferramentas para automatização da execução dos testes, ainda é necessária a execução de algumas atividades de maneira manual ou semi-automatizada como, por exemplo, a etapa de elaboração dos casos de teste. Esta geração manual ou semi-automatizada torna o processo de teste ineficiente e suscetível à inserção de falhas[3] nestes casos de teste até mesmo por profissionais experientes. Uma alternativa para melhorar o uso destas ferramentas é automatizar o processo de geração de casos de teste por meio de técnicas de teste de *software*. Uma das técnicas que auxiliam o processo de teste de *software* é o Teste Baseado em Modelos *(Model-Based Testing* - MBT) [Krishnan 2004]. Esta técnica consiste na geração dos casos de teste e/ou *scripts* de teste baseado no modelo do *software*. Modelo este que, inclui a especificação das características que serão testadas [Krishnan 2004]. Além disso, o uso de MBT apresenta várias vantagens, *e.g.* a probabilidade de redução da má interpretação dos requisitos do sistema por um engenheiro de teste e/ou a redução do tempo do teste.

Neste contexto, este artigo apresenta uma abordagem relacionada à automatização de casos de teste estrutural para diversas tecnologias a partir de diagramas de sequência UML. Esta abordagem consiste em três etapas: (a) anotação dos diagramas de sequência com as informações das classes e métodos que se deseja testar; (b) geração automática de uma estrutura abstrata de teste estrutural, que possui um formato genérico e descreve as informações do teste extraídas dos diagramas de sequência criados; (c) gerar automaticamente casos de teste para uma tecnologia de teste específica a partir das informações presentes na estrutura abstrata de teste estrutural.

Uma das vantagens desta abordagem está relacionada à capacidade de reuso das informações de teste. Assim, as informações descritas na estrutura de teste, originada a partir desta abordagem, podem ser utilizadas para gerar casos de teste para diversas tecnologias, *e.g.* acadêmicas: JaBUTi [Eler et al. 2009], Poke-Tool [Chaim 1991]; comerciais [Yang et al. 2006]: Semantic Designs Test Coverage, IBM Rational PurifyPlus. Desta forma, uma empresa pode se adaptar mais rapidamente a uma eventual migração de tecnologia de teste, pois devido ao fato de seus engenheiros de teste não terem a necessidade de criar casos de teste manualmente, não necessitam de um conhecimento aprofundado referente à nova tecnologia. Outra vantagem de nossa abordagem está relacionada à utilização de modelos UML para a geração e execução de casos de teste. Modelos provêem uma representação das informações de teste em alto nível, facilitando o entendimento por parte do analista, ou dos responsáveis pela implementação e execução dos casos de teste.

Com base nesta abordagem foi desenvolvida uma ferramenta capaz de gerar de forma automatizada a estrutura abstrata a partir de diagramas de sequência e, instanciar as informações presentes nesta estrutura para automaticamente gerar e executar casos de teste concretos para uma tecnologia de teste específica, a JaBUTi. É importante ressaltar que a ferramenta apresentada neste artigo foi gerada a partir de uma linha de produto denominada PLeTs PL [Rodrigues et al. 2010]. A PLeTs PL é uma linha de produto de ferramentas de teste capaz de gerar produtos (ferramentas) que automatizam a geração e execução de casos de teste aplicando MBT.

[3]Neste artigo utilizamos falha (*fault*), erro (*error*) e defeito (*failure*) segundo [Avizienis et al. 2004].

Este artigo está estruturado da seguinte forma: a Seção 2 faz uma breve introdução dos conceitos de teste baseado em modelos, teste estrutural e linha de produto de *software*. A Seção 3 apresenta detalhadamente a abordagem deste trabalho, bem como as funcionalidades da ferramenta que implementa a abordagem descrita. A Seção 4 apresenta a definição de alguns casos de teste para a nossa ferramenta utilizando uma aplicação que gerencia habilidades de uma empresa em uma empresa de TI como exemplo de uso. Ao final, a Seção 5 apresenta as conclusões e trabalhos futuros.

2. Fundamentação Teórica

Nesta seção serão apresentados os conceitos relativos a teste baseado em modelos, teste estrutural e linha de produto de *software*. Além disso, será apresentada a descrição da linha de produto de *software* que está diretamente relacionada com o tema abordado neste trabalho, *i.e.* PLeTs PL [Rodrigues et al. 2010].

2.1. Teste Baseado em Modelos

Teste Baseado em Modelos (*Model-Based Testing* - MBT) é uma técnica para a geração automática de artefatos de teste com base em modelos extraídos de artefatos de *software* [Dalal et al. 1999]. O principal objetivo de MBT é a criação de artefatos de teste que descrevam os requisitos e comportamento do próprio sistema, visando automação do processo de teste de *software*. Assim, o teste de *software* baseado em MBT requer a elaboração de modelos de teste para determinar parte de esforço do teste. Engenheiros de teste, quando elaboram os casos de testes do sistema, implicitamente, constroem estes modelos mentalmente [Sarma et al. 2010].

Modelos de teste possibilitam encapsular o comportamento e a estrutura do sistema, a fim de que sejam compartilhados e reutilizados pela equipe de teste. Desta forma, a partir destes modelos, é possível extrair as informações anotadas para a geração de novos artefatos de teste, tais como casos de teste, *scripts* e cenários de teste [El-Far and Whittaker 2001]. Uma vez desenvolvido o modelo, ele pode ser utilizado de várias formas e por várias etapas do processo de desenvolvimento de *software*, tais como: especificação dos requisitos, geração de código, análise de confiabilidade do sistema e execução de testes [Broy et al. 2005].

O custo do teste de *software* está relacionado ao número de interações e casos de teste que são executados durante o processo de desenvolvimento. Como ele é um dos processos mais onerosos e caros do desenvolvimento de *software* [Krishnan 2004], MBT é uma ótima abordagem para mitigar este problema, automatizando o processo de geração de casos de teste ou *scripts* com o objetivo de reduzir tempo e custo do processo de teste. Entretanto, no contexto da automatização da geração de casos de teste raramente empresas adotam MBT em suas atividades, permanecendo a maioria delas no processo manual do teste [Sarma et al. 2010].

A adoção da abordagem MBT exige a criação de modelos formais baseados nos requisitos especificados por engenheiros e analistas de testes. O objetivo é que estes modelos possam detectar informações que na maioria das vezes estão implícitas em documentos tradicionais de especificação, anotando comentários e estereótipos ao modelo que enriquecem a qualidade da especificação. Desta forma, estas informações incrementadas ao modelo servirão para a criação de novos artefatos, ou ainda, permitindo a

automatização de outros processos para melhorar a comunicação da equipe e a qualidade dos artefatos desenvolvidos. Esta modelagem pode ser implementada em vários contextos, pois a abordagem de MBT se aplica a diversos tipos e/ou níveis de teste de *software* [Dias Neto et al. 2007].

Os primeiros estudos que abordam modelagem de teste de *software* se limitavam à técnica funcional, *i.e.* teste caixa preta, descrevendo os aspectos funcionais a serem testados. Atualmente, os modelos são capazes de abstrair inúmeras outras informações, *e.g.* parâmetros e dados de entrada do teste, permitindo ser aplicada a técnica MBT para realização de outros tipos de testes, *e.g.* o teste estrutural [Delamaro et al. 2007].

2.2. Teste Estrutural

O teste estrutural é uma técnica que tem por objetivo a geração de casos de teste a partir da análise do código fonte. Busca avaliar os detalhes internos da implementação, tais como teste de condição, caminhos lógicos, etc. Por este motivo, é também chamado de teste orientado à lógica ou teste caixa-branca (*white-box*) [Delamaro et al. 2007]. Como citado anteriormente, a técnica de teste estrutural se baseia na análise da estrutura do programa para a geração dos casos de teste. Esta técnica define um conjunto de critérios estruturais, os quais se baseiam em diferentes elementos de um programa para definir requisitos de teste. Segundo [Delamaro et al. 2007], estes critérios tem por objetivo a execução de componentes e partes elementares de um programa e, são classificados basicamente em critérios baseados no(a):

- *Fluxo de controle:* utiliza análise do fluxo de controle do programa para gerar casos de teste. Utiliza os aspectos de controle do programa, tais como, laços, desvios ou condição para derivar os casos de teste;
- *Fluxo de dados:* utiliza análise do fluxo de dados do programa para gerar casos de teste. Casos de testes são derivados a partir de associações entre definições de variáveis e o uso dessas variáveis;
- *Complexidade:* fundamentam-se em informações de complexidade do programa para levantamento dos requisitos de teste. O critério de teste de caminho é um dos mais conhecidos critérios de complexidade.

Todavia, a maioria dos testes é baseada nas especificações do *software*, o que não é o caso do teste estrutural, em que são avaliados os diferentes caminhos que o programa pode executar, a partir de sua implementação. Muitas vezes, apesar de testados os possíveis caminhos mapeados de execuções de um programa, isso não é garantia de que o programa não possa vir a apresentar defeitos, pois algum caminho pode não ter sido percorrido. E ainda, a execução de um comando ou programa com falha, infelizmente, pode não resultar na geração de defeitos percebidos pelo usuário final [Pezzè and Young 2008]. Entretanto, o teste estrutural é relevante para fatores de qualidade do *software* como manutenabilidade, estrutura e confiança, pois inclui casos de teste que não são avaliados através de testes funcionais, *e.g.* identificar se todo o código foi executado.

2.3. Linha de Produto de *Software*

Uma Linha de Produto de *Software* (*Software Product Line - SPL*) é definida como um conjunto de sistemas que compartilham características comuns e gerenciáveis com o intuito de satisfazer as necessidades de um domínio específico, podendo este ser um segmento de mercado ou missão [Clements and Northrop 2001]. O objetivo é explorar as

semelhanças entre os sistemas visando gerenciar os aspectos relativos à variabilidade entre eles e, dessa forma, determinar uma maior reusabilidade dos componentes de *software*. Por meio desta capacidade de reutilização de componentes, uma SPL possibilita criar um conjunto de sistemas similares, reduzindo assim o tempo de comercialização, custo e com isso, obter maior produtividade e melhoria da qualidade.

A parte mais importante de uma SPL diz respeito ao núcleo de artefatos, o qual forma a base de uma SPL e pode ser formado por componentes reusáveis, modelos de domínios, requisitos da SPL, casos de teste e modelo de características (*feature models*). O modelo de características apresenta todas as características de uma linha de produto e a relação entre os componentes. O modelo de características de uma SPL é responsável por representar os aspectos relacionados à variabilidade, a qual pode estar vinculada a diferentes níveis de abstração como código fonte e documentação. As variabilidades são representadas por pontos de variabilidades (*variation points*) e variantes (*variants*), onde um ponto de variabilidade pode conter uma ou mais variantes. Em uma linha de produto de telefones móveis, por exemplo, um ponto de variabilidade poderia ser o protocolo de comunicação e suas variantes poderiam ser GSM (*Global System for Mobile Communications*) ou UMTS (*Universal Mobile Telecommunication System*).

No contexto de teste, atualmente, está em desenvolvimento no Centro de Pesquisa em Engenharia de Sistemas da PUCRS uma linha de produto denominada PLeTs PL [Rodrigues et al. 2010]. A PLeTs PL é uma SPL que busca facilitar a derivação de ferramentas de teste baseado em modelos, com os quais é possível criar e executar casos de teste de forma automatizada. O objetivo desta SPL não é apenas gerenciar a reutilização de artefatos e componentes de *software*, mas também tornar mais fácil e rápido o desenvolvimento de uma nova ferramenta que, como citado anteriormente, tem por objetivo otimizar a criação e execução de casos de teste. Para melhor entendimento, é apresentado na Figura 1 o modelo de características atual da PLeTs PL. Atualmente, o modelo de características da PLeTs PL é constituído de quatro características em seu primeiro nível:

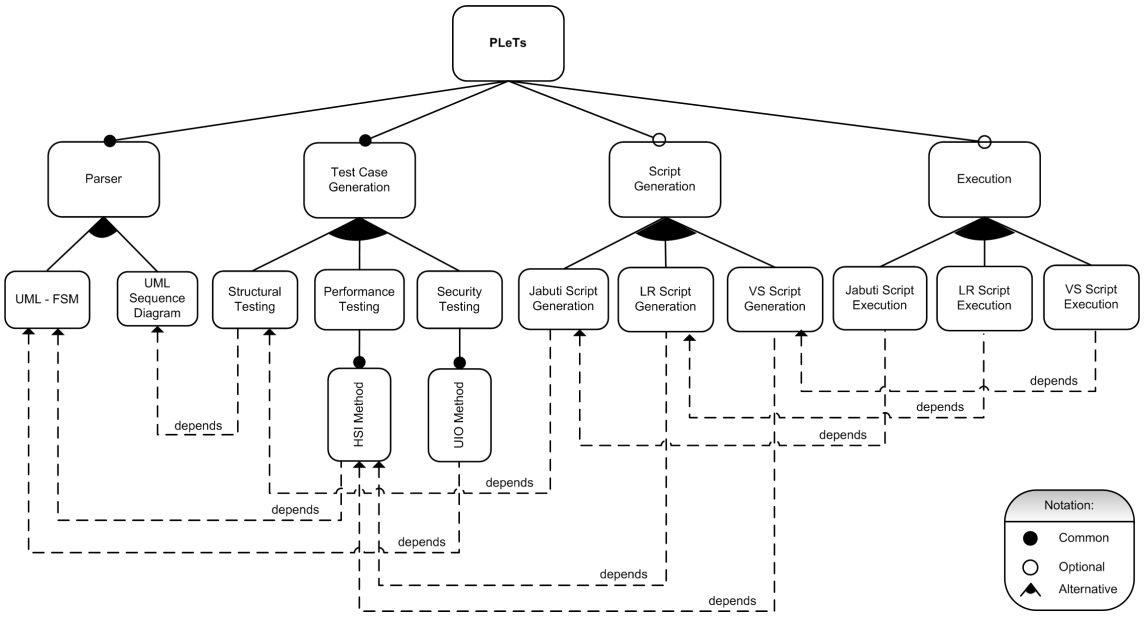


Figura 1. Modelo de características da PLeTs PL [Rodrigues et al. 2010]

1) *Parser*: esta é uma característica ou ponto de variabilidade obrigatório e seu objetivo é extrair as informações contidas, por exemplo, em um arquivo UML (Diagrama de Sequência, Diagrama de Atividades) para então gerar um modelo formal na característica `Test Case Generation`. Atualmente, esta característica possui duas variantes mutuamente excludentes: `UML - FSM (Finite State Machine)` e `UML - Sequence Diagram`, onde apenas uma das duas variantes é selecionada; 2) *Test Case Generation*: também é uma característica obrigatória. É responsável por gerar sequências de teste a partir das informações contidas no modelo formal. Podem ser geradas sequências para teste estrutural, desempenho ou segurança. Este ponto de variabilidade possui três variantes (`Structural Testing`, `Performance Testing` e `Security Testing`) e obrigatoriamente, assim como na característica `Parser`, apenas uma delas pode ser selecionada. Como pode-se observar na Figura 1, as três variantes possuem uma relação de dependência (*depends*) com as variantes do `Parser`. Indicando que a seleção de uma variante é dependente de outra. Por exemplo, caso a variante `Structural Testing` seja escolhida, obrigatoriamente a variante `UML - Sequence Diagram` deve ser selecionada; 3) *Script Generation*: consiste na geração de *scripts* de teste para ferramentas de automação de teste. Também possui três variantes, sendo elas: `Jabuti Script Generation`, `LR Script Generation` e `VS Script Generation`, as quais representam a implementação de *scripts* e cenários de teste[4], respectivamente, para as ferramentas JaBUTi, LoadRunner e Visual Studio; 4) *Execution*: representa a execução da ferramenta e também a execução do teste sob a aplicação a ser testada, utilizando uma determinada ferramenta com os *scripts* e cenário de teste gerados na etapa anterior. Possui três variantes e cada uma possui uma relação de dependência com as variantes do ponto de variabilidade `Script Generation`.

A relação da PLeTs PL com este trabalho diz respeito ao fato de que um dos produtos gerados por esta SPL refere-se à ferramenta abordada neste artigo. Portanto, nas seções seguintes serão apresentadas de forma detalhada as funcionalidades desta ferramenta, bem como a abordagem que a originou.

3. Geração de Casos de Teste Estrutural

Em trabalhos anteriores [Silveira et al. 2011] [Rodrigues et al. 2010] foram apresentadas abordagens com as quais é possível automatizar a geração e execução de casos de teste de desempenho de *software*. Nesse trabalho será apresentada a aplicação de uma abordagem que, diferentemente das anteriores, tem por objetivo automatizar a geração e execução de casos de teste estrutural a partir de diagramas de sequência.

A aplicação de nossa abordagem é dividida em três etapas (ver Figura 2). Inicialmente, se faz necessária a anotação do diagrama de sequência que descreve as classes e métodos da aplicação que se deseja testar (ver Figura 2 - a). É importante que este diagrama esteja "completo", contendo informações referentes aos parâmetros (nome, tipo) destas classes e métodos, bem como o tipo de retorno de cada método. Caso estas informações não estejam explícitas, o diagrama deve ser anotado adicionando tais informações. Desta forma, o próximo passo diz respeito à anotação deste diagrama com informações que serão usadas para a etapa seguinte de nossa abordagem (maiores detalhes a respeito de como o diagrama é anotado serão descritos na Seção 4). Para anotar

[4]Cenários de teste contém um ou mais *scripts* e possuem informações referentes à configuração do teste.

estas informações é utilizado o seguinte conjunto de *tags*:

<<**TDexternalLibray**>>: especifica o caminho das bibliotecas da aplicação sob teste;
<<**TDclassPath**>>: especifica o caminho das classes que serão testadas;
<<**TDtechnologyPath**>>: especifica informações particulares à tecnologia de teste escolhida, *e.g.*, diretório de instalação e o caminho de seu arquivo executável;
<<**TDimportList**>>: especifica uma lista de classes importadas (*import, package*).

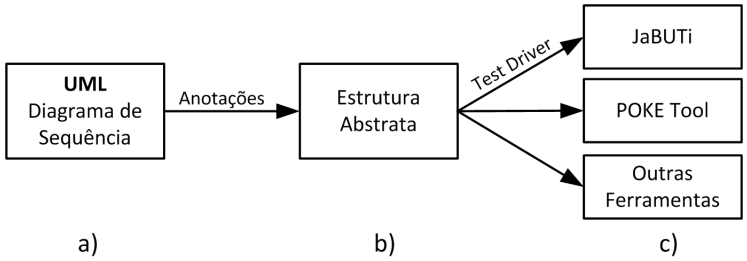


Figura 2. Abordagem para a geração de casos de testes estrutural

Após este processo de anotação do diagrama, é gerado um arquivo no formato *XMI* contendo as informações descritas no diagrama de sequência anotado. A segunda etapa de nossa abordagem consiste em submeter o arquivo *XMI* a um *parser*, o qual é responsável por extrair os dados presentes neste arquivo para então gerar uma estrutura abstrata de teste (ver Figura 2 - b). Esta estrutura, a qual chamamos de `Estrutura Abstrata`, se trata de um arquivo de texto que descreve em um formato genérico e sequencial todo o fluxo de dados das classes e métodos da aplicação que se deseja testar.

A vantagem de utilizar esta estrutura está relacionada à capacidade de reuso das informações nela descritas por diversas tecnologias de execução de teste estrutural, *e.g.* JaBUTi [Eler et al. 2009], Poke-Tool [Chaim 1991]. Neste sentido, caso uma empresa de TI decida, devido a uma estratégia administrativa, migrar para uma tecnologia de teste estrutural diferente da que possui, poderá aproveitar os casos de teste previamente definidos. Além de facilitar uma migração de tecnologia, a estrutura abstrata apresenta as informações do teste em um formato claro, tornando simples e fácil o entendimento dos casos de teste definidos.

A última etapa de nossa abordagem diz respeito à instanciação da estrutura abstrata para determinada tecnologia e a configuração de informações particulares à tecnologia de teste escolhida (ver Figura 2 - c). Esta etapa será descrita com maiores detalhes nas próximas seções, onde será apresentado um exemplo real com a estrutura abstrata sendo instanciada para gerar casos de teste para determinada aplicação. Ainda que nossa abordagem possa suportar diversas tecnologias, para este trabalho foi utilizada a ferramenta JaBUTi. Portanto, será apresentada a nossa ferramenta, a qual é capaz de gerar e executar automaticamente casos de teste para a JaBUTi a partir de informações descritas em diagramas de sequência.

3.1. Geração de Casos de Teste para a JaBUTi

Desenvolvida pelo grupo de Engenharia de *Software* da USP-São Carlos, a JaBUTi [Eler et al. 2009] (*Java Bytecode Understanding and Testing*) é uma ferramenta utilizada para execução de teste estrutural em aplicações Java. Ao contrário de outras ferramentas de teste estrutural, a JaBUTi é capaz de executar testes sem que seja necessária uma

análise do código fonte, pois a execução de testes pela ferramenta é realizada a partir da análise do Java *Bytecode*. Para a análise de cobertura das classes e componentes dos programas a serem testados, a Jabuti se baseia em oito critérios de teste estrutural, sendo quatro critérios de análise de fluxo de dados e outros quatro para análise de fluxo de controle [Vincenzi. et al. 2005].

Para tornar possível a geração automática de casos de teste com a JaBUTi foi necessária uma análise para entender o funcionamento interno da ferramenta. Para a execução de casos de teste com a JaBUTi é necessário criar um arquivo de projeto, cuja extensão é `.jbt`. Todas as informações referentes ao *bytecode* das classes que serão testadas e o caminho dessas classes são armazenadas neste arquivo. Neste arquivo também são descritos os caminhos referentes a todas as bibliotecas que pertencem à aplicação a ser testada, bem como o *bytecode* da classe `TestDriver`, que contém informações que serão utilizadas para testar as classes e métodos da aplicação sob teste. Com base nas informações do *bytecode* das classes, a JaBUTi constrói o grafo definição-uso (*Def-Use Graph* - DUG) para cada uma dessas classes.

Após a criação do arquivo de projeto, a JaBUTi realiza a instrumentação das classes a serem testadas. Ela executa os casos de teste, descritos no arquivo `TestDriver.java`, por meio da chamada do método `probe.DefaultProber.probe` que armazena as informações do programa que será testado. Em seguida, outro método é chamado (`probe.DefaultProber.dump`) e todos os dados coletados na chamada do método anterior são armazenados em um arquivo de rastro (`.trc`). Os dados gravados no arquivo de rastro correspondem aos caminhos percorridos pelo programa durante a execução do teste. A JaBUTi extrai as informações deste arquivo de rastro, atualiza os dados do teste e recalcula as informações de cobertura.

De posse desse conhecimento, foi possível verificar que nossa ferramenta é capaz de automatizar a geração e execução de casos de teste com a JaBUTi por meio da criação de dois arquivos. Estes arquivos se tratam da `TestDriver.java` e do arquivo de projeto da ferramenta, sendo que ambos são gerados automaticamente a partir das informações contidas na estrutura abstrata. Uma vez que estes dois arquivos são gerados, a execução dos casos de teste consiste na chamada interna dos métodos `probe.DefaultProber.probe` e `probe.DefaultProber.dump`. Ao final, a nossa ferramenta cria um processo Java para executar a JaBUTi contendo as informações de cobertura já atualizadas.

4. Exemplo de Uso

Nesta seção será apresentada a definição de um conjunto de casos de teste para serem gerados e executados a partir da aplicação de nossa abordagem. Para isto, será utilizada como exemplo de uso uma aplicação denominada Skills, a qual tem por objetivo a gerência de perfis profissionais de funcionários de uma empresa. O objetivo é verificar e avaliar os aspectos estruturais a partir da ferramenta que implementa a abordagem apresentada na seção anterior.

4.1. Definição dos Casos de Teste

A aplicação utilizada como exemplo de uso para este trabalho é denominada Skills (*Workforce Planning: Skill Management Prototype Tool*) [Silveira et al. 2011] e tem por obje-

tivo gerenciar os perfis profissionais de funcionários de uma dada empresa. Esta aplicação foi desenvolvida por um grupo de pesquisa da PUCRS em colaboração com uma empresa de tecnologia e tem como principal funcionalidade o gerenciamento do cadastro de habilidades, certificações e experiências de funcionários.

Com o intuito de verificar os aspectos funcionais de nossa abordagem, foi definido um caso de teste para ser executado pela nossa ferramenta. Para isso, foi definido um caso de teste para a análise estrutural de alguns métodos da aplicação Skills. O caso de teste definido descreve o processo referente à busca de informações de determinada certificação por parte do usuário. As informações referentes ao caso de teste são representadas por um diagrama de sequência, o qual foi definido e implementado de forma manual (ver Figura 3). Como é possível visualizar neste diagrama, a interação do usuário com a aplicação inicia por meio da inserção de informações referentes ao nome da certificação e do fornecedor (`provider`). Em seguida, são chamados três métodos pertencentes à classe `ServletCertification`.

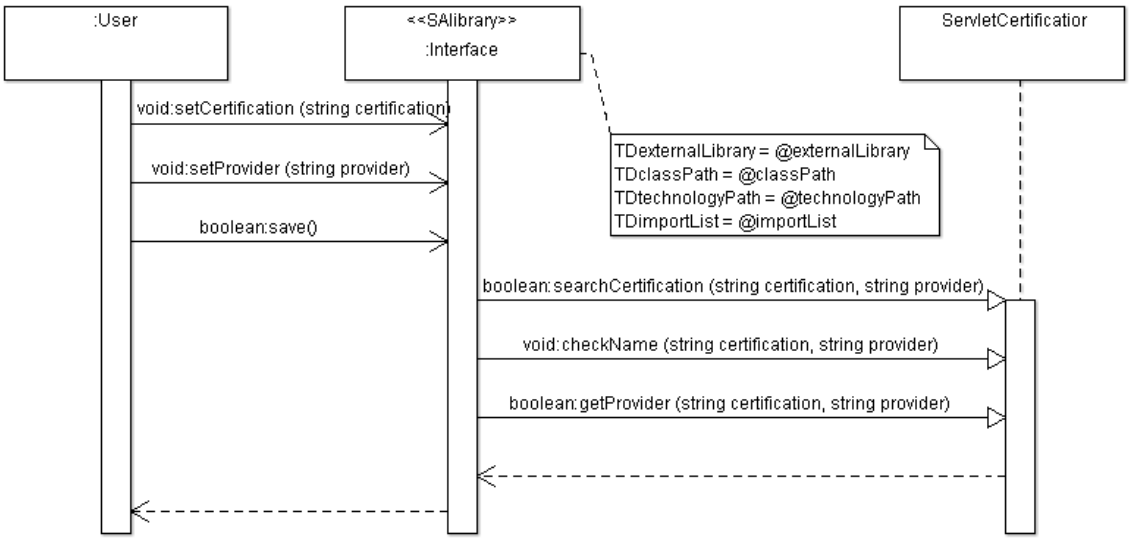


Figura 3. Diagrama de sequência para busca de certificação

O primeiro método (`searchCertification`) é responsável por acessar a base de dados e retorna se a certificação e `provider` informados pelo usuário foram cadastrados no sistema. O segundo método (`checkName`) faz a concatenação dos nomes da certificação e `provider`, enquanto o terceiro método (`getProvider`) verifica a existência de um `provider` dado um nome de certificação que é também composto pelo nome do `provider` (por exemplo, procura-se a certificação `Certified Scrum Master` do `provider` `CSM`, quando, na realidade, o nome da certificação é `CSM - Certified Scrum Master`). Ao final, são apresentadas ao usuário as informações referentes à data de aquisição e descrição da certificação informada previamente por ele.

Conforme descrito na Seção 3, o diagrama de sequência pode ser anotado com quatro *tags*: `TDexternalLibray`, `TDclassPath`, `TDtechnologyPath` e `TDimportList`. Para o nosso exemplo, estas quatro *tags* foram anotadas no diagrama de sequência, sendo que cada uma possui seu respectivo parâmetro, `@externalLibrary`, `@classPath`, `@technologyPath` e `@importList`. A

partir do diagrama de sequência é gerado um arquivo no formato *XMI*, o qual servirá como entrada para a execução de nossa ferramenta. Durante sua execução, a ferramenta submete o arquivo *XMI* a um *parser*, extraindo as informações dos métodos e da classe a serem testados para gerar a estrutura abstrata.

É importante destacar que nossa abordagem consiste na geração e execução automatizada de casos de teste onde somente são analisados os métodos das classes internas do sistema. Portanto, nenhum método chamado a partir da interação do usuário será analisado. Neste contexto, somente as informações referentes aos métodos `searchCertification`, `checkName` e `getProvider` da classe `ServletCertification` serão utilizadas para gerar automaticamente a estrutura abstrata. Como pode ser visto na Figura 4, a estrutura abstrata possui informações utilizadas para a geração de casos de teste, sendo dividida em três grupos: a) *Configuração da Tecnologia*: caracteriza o parâmetro `@technologyPath`, o qual especifica as informações da tecnologia a ser utilizada para o teste; b) *Configuração do Teste*: caracteriza os parâmetros `TDclassPath`, `@externalLibrary` e `@importList`, os quais definem as informações utilizadas para um caso de teste específico; c) *Configuração do Fluxo Sequencial*: caracteriza o fluxo sequencial dos métodos e classe a serem testados;

```
Estrutura abstrata: Busca Certificação
## Configuração da Tecnologia ##
Informações da Tecnologia : <<TDtechnologyPath: @technologyPath>>

## Configuração do Teste ##
Bibliotecas Externas : <<TDexternalLibray: @externalLibrary>>
Caminho das Classes : <<TDclassPath: @classPath>>
Classes Importadas  : <<TDimportList: @importList>>

## Configuração do Fluxo Sequencial ##
1. ServletCertification
1.1. searchCertification(string certification, string provider): boolean
1.2. checkName(string certification, string provider): void
1.3. getProvider(string certification, string provider): boolean
```

Figura 4. Estrutura abstrata

Cada parâmetro possui uma referência para um arquivo de dados contendo informações dos valores que serão utilizados para instanciar os casos de teste para determinada tecnologia. Portanto, quando a estrutura abstrata for instanciada para gerar os casos de teste para a JaBUTi, a classe `TestDriver.java` e o arquivo de projeto (`.jbt`) da ferramenta serão gerados com os valores descritos neste arquivo de dados. Este mesmo arquivo de dados também descreve um conjunto de informações que serão utilizadas como entrada dos métodos a serem testados. Na abordagem proposta nesse trabalho, a geração das informações destes dados de entrada é realizada, automaticamente, aplicando a técnica de geração aleatória de dados [Hamlet and Taylor 1990] a partir de um domínio finito de entradas. Esta técnica foi utilizada devido a sua praticidade, por ser menos custosa e mais simples de automatizar. Contudo, outras técnicas de geração de dados de teste são apresentadas na literatura, *e.g.* geração com execução simbólica [Lin et al. 2011] e geração com execução dinâmica [Dara et al. 2009].

Quanto à utilização de um arquivo de dados, uma de suas vantagens refere-se ao fato de que não é necessário alterar no diagrama de sequência, os valores dos parâmetros dos métodos a serem testados. Desta forma, para gerar casos de teste com diferentes

valores de entrada para os parâmetros dos métodos basta aplicar novamente a técnica de geração aleatória de dados de teste. Para exemplificar, a Figura 5 possui algumas informações referentes aos valores dos parâmetros presentes neste arquivo.

```
@technologyPath = C:\Jabuti\bin; C:\Jabuti\lib\bcel-5.2.jar; C:\Jabuti\lib\capi.jar; ...
@externalLibrary = C:\Apache Software Foundation\Tomcat 6.0\lib\jsp-api.jar; ...
@classPath = C:\Temp\Workspace\CmTool_SkillsTest\web\WEB-INF\classes; ...
@importList = servlets.*; java.io.*; java.util.StringTokenizer
1. ServletCertification
1.1. searchCertification("ActiveX", "BrainBench")
1.2. checkName("ActiveX", "BrainBench")
1.3. getProvider("ActiveX", "BrainBench")
```

Figura 5. Arquivo de dados

A partir das informações descritas na estrutura abstrata, são gerados o arquivo de projeto da JaBUTi e a classe `TestDriver.java`. Para gerar o arquivo de projeto, nossa ferramenta cria um processo Java que executa a classe `br.jabuti.cmdtool.CreateProject` da JaBUTi, onde as informações de tecnologia descritas no arquivo de dados são utilizadas como parâmetro para a execução desta classe. Por outro lado, a criação do arquivo `TestDriver.java` é menos complexa, sendo gerada a partir da estrutura abstrata com os valores extraídos do arquivo de dados. Entretanto, a `TestDriver.java` necessita ser compilada, pois como citado anteriormente a JaBUTi realiza a análise estrutural apenas a partir do *bytecode* das classes a serem testadas. Portanto, nossa ferramenta cria um processo Java para compilar este arquivo. Para exemplificar, as Figuras 6 e 7 apresentam, respectivamente, o arquivo `TestDriver.java` e o arquivo de projeto da JaBUTi.

```
import servlets.*;
import java.io.*;
import java.util.StringTokenizer;

public class TestDriver {
  static public void main(String args[]) throws Exception {
    ServletCertification servletcertification = new ServletCertification();
    servletcertification.searchCertification("ActiveX", "BrainBench");
    servletcertification.getProvider("ActiveX", "BrainBench");
    servletcertification.checkName("ActiveX", "BrainBench");
  }
}
```

Figura 6. Arquivo TestDriver.java

Ao final, a nossa ferramenta cria um processo Java para executar a JaBUTi, a qual é responsável por calcular e apresentar as informações de cobertura para o caso de teste definido. Na Figura 8 é apresentada a interface da JaBUTi com as informações dos percentuais de cobertura dos métodos testados, com base no critério estrutural `Todos os Nodos Primários` (*All-Nodes-ei*). De acordo com os resultados obtidos, os métodos `searchCertification`, `checkName` e `getProvider` foram, respectivamente, cobertos com percentuais de 52%, 100% e 38%. Como pode-se observar, os métodos `searchCertification` e `getProvider` não foram totalmente cobertos. Para aumentar o percentual de cobertura deste métodos diferentes entradas devem ser geradas.

```
<JABUTI>
    <PROJECT name="C:\Jabuti\JabutiProjects\Skills\Skills.jbt" type="research" mobility=
        "N" CFGOption="1">
        <BASE_CLASS name="TestDriver"/>
        <CLASSPATH path=". C?\Jabuti\bin C?\Temp\Workspace\CmTool_SkillsTest\web\WEB-INF
            \classes C?\Jabuti\lib\bcel-5.2.jar C?\Jabuti\lib\capi.jar C?\Jabuti\lib\
            junit-4.4.jar C?\Jabuti\lib\mucode.jar"/>
        <JUNIT_SRC_DIR dir=""/>
        <JUNIT_BIN_DIR dir=""/>
        <JUNIT_TEST_SET name=""/>
        <JUNIT_JAR name=""/>
        <AVOIDED_PACKAGES>
        </AVOIDED_PACKAGES>
        <CLASS name="TestDriver" size="0000" checksum="00000000">
            <EXTEND name="java.lang.Object" level="1"/>
        </CLASS>
        <INST_CLASS name="servlets.ServletCertification" size="9250" checksum="50-0">
            <EXTEND name="javax.servlet.http.HttpServlet" level="1"/>
            <SOURCE name="ServletCertification.java"/>
            <METHOD id="1" name="searchCertification(Ljava/lang/String;Ljava/lang/...)">
            ...
    </PROJECT>
</JABUTI>
```

Figura 7. Arquivo de projeto (`.jbt`)

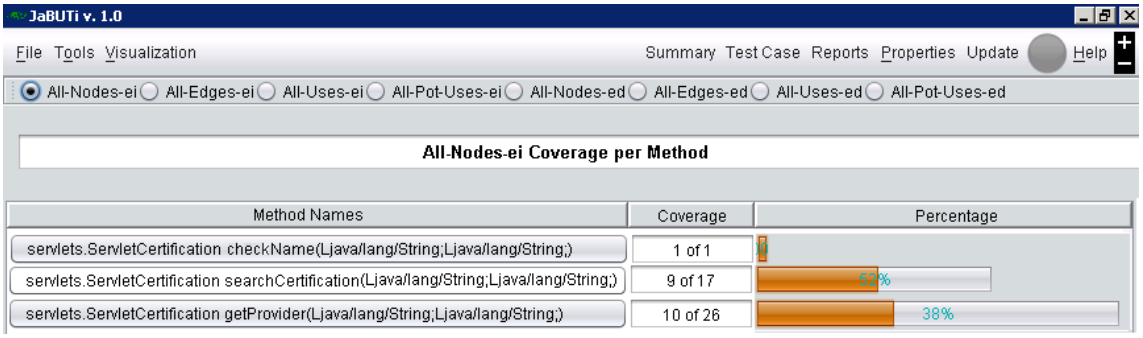


Figura 8. Interface da JaBUTi com as informações de cobertura dos métodos testados

5. Conclusão e Trabalhos Futuros

Este trabalho apresentou uma abordagem para automatizar o processo de geração e execução de casos de teste estrutural para diversas tecnologias a partir de diagramas de sequência UML. Com base nesta abordagem foi gerada uma ferramenta a partir de uma linha de produto de *software* que automatiza este processo de teste utilizando a JaBUTi.

Uma das vantagens em utilizar esta abordagem refere-se à capacidade de reuso das informações do teste descritas nos diagramas de sequência. Desta forma, uma empresa de TI pode facilmente migrar de uma tecnologia de teste para outra e aproveitar os casos de teste previamente definidos. A tecnologia de teste utilizada para exemplificar nossa abordagem foi a JaBUTi, porém tecnologias utilizadas na indústria como, por exemplo, Semantic Designs Test Coverage, IBM Rational PurifyPlus e também tecnologias acadêmicas como a Poke-Tool poderiam ser utilizadas para este fim. A Poke-Tool é capaz de prover análise estrutural de programas implementados na linguagem C++. Esta abordagem não seria aplicada para a análise de programas escritos essencialmente em C, pois ela se baseia na extração de informações de teste a partir de diagramas de sequência,

necessitando que os programas a serem testados tenham sido desenvolvidos utilizando o paradigma de programação orientada a objetos.

Outra vantagem de nossa abordagem refere-se à capacidade de integração de diferentes métodos para a geração de dados de teste. Desta forma, é possível realizar uma análise estrutural mais eficaz e que satisfaça os critérios estruturais mais relevantes a serem avaliados para o conjunto de classes e métodos que se deseja testar.

Apesar das vantagens adquiridas com a utilização de nossa abordagem, existem algumas questões em aberto e pontos que poderiam ser aprimorados. Ainda que nossa abordagem possibilite automatizar a geração e execução de casos de teste para diversas tecnologias de teste estrutural, é necessário um conhecimento aprofundado relacionado à tecnologia de automatização de teste que se deseja utilizar. É necessário conhecer os arquivos de configuração de teste da ferramenta e como as informações para o teste estão estruturadas nestes arquivos. Além disso, nossa abordagem poderia implementar outras técnicas de geração de dados de teste, *e.g.* geração com execução simbólica e geração com execução dinâmica, pois são mais eficazes e garantem a seleção de dados com maior probabilidade de revelar a existência de falhas.

Outra questão em aberto diz respeito à falta de informações de análise de resultados do teste. A abordagem apresentada permite gerar e executar casos de teste de forma automatizada, mas não dispõe de informações que podem contribuir na automatização da análise dos resultados. Por exemplo, esta abordagem poderia descrever quais as informações do teste são mais relevantes de serem consideradas para análise e qual relação entre esta ou aquela informação poderia contribuir em uma análise mais criteriosa do teste. Neste contexto, seria interessante que nossa ferramenta implementasse funções que desempenhassem o papel de oráculos, apresentando informações do teste relativas ao cumprimento ou não dos requisitos de teste especificados.

6. Agradecimentos

Avelino F. Zorzo possui bolsa de produtividade CNPq/Brasil. Elder M. Rodrigues possui bolsa CAPES/INCT-SEC. Agradecemos a DELL pelo apoio no desenvolvimento deste trabalho. Agradecemos também aos revisores do artigo.

Referências

Avizienis, A., Laprie, J. C., Randell, B., and Landwehr, C. (2004). Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transaction on Dependable Secure Computing*, 1:11–33.

Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., and Pretschner, A. (2005). *Model-Based Testing of Reactive Systems: Advanced Lectures*. Springer.

Chaim, M. L. (1991). POKE-TOOL: A Tool to Support Data Flow Based Structural Test of Programs. Master's thesis, DCA/FEEC/UNICAMP, Campinas, SP, Brasil.

Clements, P. and Northrop, L. (2001). *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing.

Dalal, S. R., Jain, A., Karunanithi, N., Leaton, J. M., Lott, C. M., Patton, G. C., and Horowitz, B. M. (1999). Model-Based Testing in Practice. In *Proceedings of the International Conference on Software Engineering*, pages 285–294.

Dara, R., Li, S., Liu, W., Smith-Ghorbani, A., and Tahvildari, L. (2009). Using Dynamic Execution Data to Generate Test Cases. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 433–436.

Delamaro, M. E., Maldonado, J. C., and Jino, M. (2007). *Introdução ao Teste de Software*. Elsevier Editora.

Dias Neto, A. C., Subramanyan, R., Vieira, M., and Travassos, G. H. (2007). A Survey on Model-Based Testing Approaches: A Systematic Review. In *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies*, pages 31–36.

El-Far, I. K. and Whittaker, J. A. (2001). *Model-based Software Testing*. Wiley.

Eler, M. M., Endo, A. T., Masiero, P. C., Delamaro, M. E., Maldonado, J. C., Vincenzi, A. M. R., Chaim, M. L., and Beder, D. M. (2009). JaBUTiService: A Web Service for Structural Testing of Java Programs. In *Proceedings of the 33rd IEEE International on Software Engineering Workshop*, pages 69–76.

Halili, E. (2008). *Apache JMeter*. Packt Publishing.

Hamlet, D. and Taylor, R. (1990). Partition Testing does not Inspire Confidence (Program Testing). *IEEE Transactions on Software Engineering*, 16:1402–1411.

Krishnan, P. (2004). Uniform Descriptions for Model Based Testing. In *Proceedings of the Australian Software Engineering Conference*, pages 96–105.

Lin, M., Chen, Y., Yu, K., and Wu, G. (2011). Lazy Symbolic Execution for Test Data Generation. *IET Software*, pages 132–141.

Pezzè, M. and Young, M. (2008). *Teste e Análise de Software - Processos, Princípios e Técnicas*. John Wiley & Sons.

Rodrigues, E. M., Viccari, L. D., Zorzo, A. F., and Gimenes, I. M. (2010). PLeTs Tool - Test Automation Using Software Product Lines and Model Based Testing. In *Proceedings of the 22th International Conference on Software Engineering and Knowledge Engineering*, pages 483–488.

Sarma, M., Murthy, P. V. R., Jell, S., and Ulrich, A. (2010). Model-based testing in industry: a case study with two MBT tools. In *Proceedings of the 5th Workshop on Automation of Software Test*, pages 87–90.

Silveira, M. B., Rodrigues, E. M., Zorzo, A. F., Costa, L. T., Vieira, H. V., and de Oliveira, F. M. (2011). Generation of Scripts for Performance Testing Based on UML Models. In *Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering*, pages 258–563.

Vincenzi., A. M. R., Maldonado, J. C., Wong, W. E., and Delamaro, M. E. (2005). Coverage Testing of Java Programs and Components. *Science of Computer Programming*, 56:211–230.

Yang, Q., Li, J. J., and Weiss, D. (2006). A survey of coverage based testing tools. In *Proceedings of the 2006 international workshop on Automation of software test*, pages 99–103.

Modeling Communication Semantics for Distributed Systems in Event-B*

Fernando Luís Dotti [1] **, Leila Ribeiro** [2]

[1]Faculdade de Informática - PUCRS - Porto Alegre - Brazil
fernando.dotti@pucrs.br

[2]Instituto de Informática - UFRGS - Porto Alegre - Brazil
leila@inf.ufrgs.br

Abstract. *During the development of algorithms for distributed systems, one has to adopt clear assumptions about the semantics offered by the underlying communication platform in order to show that the algorithms under construction fulfill the expected liveness and safety properties. In this paper we propose a library of reusable formal specifications defining several classic communication semantics. The specification of each communication semantics is presented along with the proofs of the expected main properties of each model. The library was build using Event-B and properties were shown using the theorem proving approach with the Rodin system. While modeling a distributed application one can reuse models from the proposed library (by refinement or extension) without having to redo all the proofs related to the communication platform. Moreover, existing proofs can be used to show desired properties of the application.*

1. Introduction

Building dependable systems is not a trivial task and several means are employed to enhance system dependability. Fault-prevention [Laprie et al. 2004] through the use of formal specification and analysis is one of those means.

When specifying and reasoning about distributed algorithms the assumptions about the semantics of the communication platform should be clearly defined such that (i) the desired properties of the algorithm are proven considering the appropriate communication semantics; (ii) the kind of target environment for the algorithm becomes explicit. The use of formal specification and analysis methods for distributed systems is not recent. Most commonly, specification languages support classic notions of processes and synchronization. However, there are several possible communication models available to build distributed algorithms. These models emerge from the combination of basic aspects of the message passing mechanism such as one-to-one or group communication, reliable or lossy, and several possible delivery orders (e.g.: unordered, FIFO, absolute, total and causal). The specification of a distributed algorithm requires the modeling of the underlying communication model, which is a non trivial task, as well as the distributed algorithm itself. Therefore, a library of reusable formal specifications of relevant communication models is highly desirable. This is the contribution of this paper, that provides formal definitions of several communication semantics in the asynchronous model of computation, both for point-to-point as well as for group communications. More specifically, we present 3 models for point-to-point communication: unordered, unordered lossy, and ordered message delivery; and 5 for group communication: unordered, fifo, total, absolute

*This work is partially supported by FAPERGS and CNPq.

and causal orders of message delivery. The specification of each model is characterized by its main properties, that are then proven using the theorem proving approach.

Most of the approaches to formalize and reason about distributed systems are based on model-checking, which is usually more amenable to non-theoreticians if compared to theorem proving. One of the main advantages of model checking is that the process of proving properties is fully automatic. However, there are some drawbacks of this approach. One of them is the impossibility to handle infinite (or very large systems). Another relates to how model checkers are typically used. The common approach is to translate a specification/program to the input language of the model checker and then verify the properties using the tool. However, in most of the cases, there is no proof of correctness of this translation. Such proofs are usually very involved (see e.g. [Ribeiro et al. 2011]), but are absolutely necessary to ensure that the properties shown to hold by the model checker will actually hold in the actual specification or implementation.

In the approach proposed in this paper, we use the theorem proving approach, relying on the Event-B method [Abrial 2010]. Event-B is based on B [Abrial 2005], but adds the concepts of machine and events, that make the language very well-suited to the specification of distributed and concurrent systems. While analysis through theorem proving is not fully automatic, it allows to consider infinite state systems or systems with arbitrary size. The refinement technique supported by Event-B allows stepwise developing of systems, by gradually introducing implementation details. This way, the transformation step from a very concrete Event-B specification (after a series of refinements) to code in a programming language is easier to perform (and consequently less error prone). One of the potential reasons for this increase of use of Event-B is the current tool support for specification and theorem proving offered by the RODIN platform[DEPLOY Project].

This paper is organized as follows: in Section 2 we present informally the several communication models defined in the library; in Section 3 we review the Event-B formal method; in Section 4 the library of communication models is presented, starting with basic data structures, followed by models for point-to-point and group communication. Section 5 presents a discussion on how to use the proposed library, and Section 6 comments on related work, future directions and conclusions.

2. Communication Models

In this contribution we focus on message passing communication. Within this scope, we can classify communication according to the number of processes involved and the several delivery orders. In the first dimension we have point-to-point[1] or group communication; and in the second dimension we have: unordered, fifo, causal and total (the last two applicable only to group communication) [Birman 1996]. Combining these dimensions, the following communication models for point-to-point communication were modeled:

- unordered, lossy channel: communication is among a producer and a consumer part and messages can be delivered[2] out of order as well as be lost;
- unordered, reliable: same as above, but all messages are delivered;
- FIFO: all messages are delivered in the order they are created.

[1] In this text also referred to as one-to-one.

[2] In the Distributed Systems literature message “delivery” means the arrival of a message at the destination and that the message can be consumed according to the communication criteria in use.

Note that we enumerate communication models according also to their non-determinism, the former models exhibit a reacher behavior than the next ones. Regarding group communication, the following orders were modeled:

- unordered: messages are delivered at the group members in any order;
- FIFO: any two messages from the same source are delivered at each member in the order they were sent by the source. However, messages from different sources may be delivered in different orders at different members;
- causal: message delivery is according to the "happened-before" (or $\rightsquigarrow$) relationship by Lamport [Lamport 1978]; Any message m delivered at process p_s before it sends m' has to be delivered at process p_r before m' is delivered. The causal relation is transitive (if $m \rightsquigarrow m' \rightsquigarrow m''$ then $m \rightsquigarrow m''$) and partial.
- total: messages are delivered in the same order within a group. This order is not restricted to the order in which the messages were created, the important fact is that delivery is identical in all processes belonging to the same group.

3. Event-B

Event-B [Abrial 2010] is a state-based formalism closely related to Classical B [Abrial 2005].

Definition 1 (Event-B Model, Event) *An Event-B Model is defined by a tuple* $EBModel = (c, s, P, v, I, R_I, E)$ *where* c *are constants and* s *are sets;* v *are the model variables*[3]*;* $P(c, s)$ *is a collection of axioms constraining* c *and* s*;* $I(c, s, v)$ *is a model invariant limiting the possible states of* v *s.t.* $\exists c, s, v \cdot P(c, s) \wedge I(c, s, v)$ *- i.e.* P *and* I *characterise a non-empty set of model states;* $R_I(c, s, v')$ *is an initialisation action computing initial values for the model variables; and* E *is a set of model* events.

Given states v, v' *an event is a tuple* $e = (H, S)$ *where* $H(c, s, v)$ *is the guard and* $S(c, s, v, v')$ *is the before-after predicate that defines a relation between current and next states. We also denote an event guard by* $H(v)$*, the before-after predicate by* $S(v, v')$ *and the initialization action by* $R_I(v')$.

A model has a set of possible initial states, defined by the initialization action, that by construction satisfy the model's invariant I. Given a state s satisfying I, an enabled event on s leads to another state s' which also satisfies I. The existence of such an enabled event and the invariant non-violation are assured by construction trough a series of proof obligations on each event of the model. The behavior of the model is obtained by the application of every possible event to every valid state, starting from the set of states valid according to the initialization action, giving rise to a state transition system.

To refine model M we construct a new model M' that is behaviorally related to the old one. In Event-B, this is achieved by constructing a refinement mapping between M' and M and by discharging a number of refinement proof obligations.

4. Formalizing Communication Semantics using Event-B

Now we present the library of formal specifications that represent the several communication semantics listed in Section 2. They were developed in Event-B, using a refinement

[3]For convenience, as in [Abrial 2005], no distinction is made between a set of variables and a state of a system.

approach. The invariants and variants are used to characterize and prove the main properties of each of these models.

The most abstract specification, $M0$, is the less restricted one in terms of possible computations, and models basic elements of unordered reception in a lossy channel. $M1$ refines $M0$ introducing the sender part and restricting the behavior to reliable channels. $M1loss$ refines $M0$ analogously to $M1$, but allowing message loss. $M2$ refines $M1$ including an order in message generation. In this model the order is not respected in the reception, $M2$ models reliable unordered reception. $M2ordered$ refines $M2$ restricting its computations to ordered reception of messages, modeling FIFO communication.

The most abstract group communication specification, $M3$, refines $M2$ introducing: multiple processes with unique identifications; the concept of process group; message addressing using groups; and multiple message reception (at each process in the destination group). $M3$ thus presents all basic elements to model unordered group communication. $M3fifo$, $M3total$ and $M3causal$ each refine $M3$ to restrict possible computations representing, respectively FIFO, total, and causal orders in group communication.

Section 4.1 describes the data structures of the models; Sections 4.2 and 4.3 present the point-to-point and group communication models, respectively.

4.1. Basic Data Structures

The first step to construct a model using Event-B is to define the main types of elements that play important roles in the reality being modeled. These types are defined as $Sets$ in an Event-B context. Special elements of these types as well as basic operations to manipulate them can be defined as $Constants$, the behavior of the operations is restricted by stating suitable $Axioms$.

In the following, we present the main types used in our description of communication models. First, we present general structures that will be used both in point-to-point as well as in group communication, and then we list the structures that are specific to group communication. To ease understanding and save space, we will not use strictly the Event-B syntax here. We will rather list the main sets, operations and axioms together with informal explanations[4]. We also list some auxiliary theorems that were proven.

4.1.1. General Communication Structures

The basic data type in our communication models is a set of messages ($Msgs$). The intuition is that this set should contain all messages that should be delivered in a system in some period of time. We assume that this set is finite because in a any (finite) time interval, it is not possible that an infinite number of messages is delivered. The number of messages in this set is arbitrary, this is not stated in the model. We use two operations on messages: $idmsg$, that returns the message identifier (modeled by a natural number), and $next$, that, given a set of messages, returns the one with the smallest identifier. These structures can be seen in Figure 1. The operation $idmsg$ is a total injective function, guaranteeing that all messages have unique identifiers, whereas $next$ is a total function mapping each non-empty subset $msgs$ of $Msgs$ to one of its messages, this function is

[4]In Event-B, names of operations are declared as constants and their types as axioms.

defined in axiom **ax1** by using the inverse $idmsg^{-1}$ over the number that is the minimum of all numbers in the range of a function obtained by restricting $idmsg$ to those pairs whose domain is in the set $msgs$. Theorem **the1** states that the identifier of the $next$ of a group of messages is really the least one in this set. The fact that this theorem could be proven assures that our definition of the $next$ operation is correct.

SETS
$Msgs$

OPERATIONS
$idmsg$: $Msgs \rightarrowtail \mathbb{N}$
$next$: $\mathbb{P}_1(Msgs) \rightarrow Msgs$

AXIOMS
ax1 : $\forall msgs \cdot msgs \subseteq Msgs \land msgs \neq \varnothing \Rightarrow next(msgs) = idmsg^{-1}(min(ran(msgs \lhd idmsg)))$

THEOREMS
the1 : $\forall msgs \cdot \forall m \cdot msgs \subseteq Msgs \land m \in msgs \Rightarrow (m \neq next(msgs) \Rightarrow idmsg(m) > idmsg(next(msgs)))$

Figure 1. Basic Data Structures

4.1.2. Group Communication Structures

To specify group communication, additionally to the set of messages we define the sets of processes ($Proc$) and groups ($Groups$). These sets are also required to be finite. We define 4 operations: $idproc$, that assigns a unique identifier to each process (a natural number); $source$, that assigns a source process to each message; $target$, that assigns a target group to each message; and $members$, that assigns a set of processes to each group. These structures can be visualized in Figure 2.

SETS
$Procs$, $Groups$

OPERATIONS
$idproc$: $Procs \rightarrowtail \mathbb{N}$
$target$: $Msgs \rightarrow Groups$
$source$: $Msgs \rightarrow Procs$
$members$: $Groups \rightarrow \mathbb{P}_1(Procs)$

Figure 2. Group Data Structures

4.2. Point-to-Point Communication

The behavior of each communication model will be described by corresponding Event-B machines. A machine is defined by a set of variables, describing states; a set of invariants, used to define the types of variables and state properties that should remain the same throughout execution; a variant, that is an expression used to prove termination; and a set of events, that actually define which state changes may occur. In the following, we start with a very abstract machine, representing a system that might behave well, but might also lose messages, and refine it further until we get a model in which we can guarantee that all messages arrive in the same order they were generated.

Again, we will not follow strictly the Event-B syntax. We will describe the set of variables of each machine, together with the corresponding types (instead of declaring the types using invariants), the relevant invariants/variant and all events. The initialization event will be omited because it is typically trivial.

4.2.1. Model M0

We start with a very simple machine with one variable ($deliveredMsgs$) that describes the set of messages that have been successfully delivered. This set is of course a subset of the set of all messages that should be delivered ($Msgs$). The guard of the *RECEIVEor-LOSE* event assures that it may only happen if there is a message m that has not yet been delivered. The action is to update the set of delivered messages. However, at this abstract level, the action is a non-deterministic assignment modeling the fact that this message may either be delivered or not. Figure 3 presents this model.

MACHINE *M0*
VARIABLES
$deliveredMsgs \subseteq Msgs$

EVENTS
Event *RECEIVEorLOSE* $\widehat{=}$
any m
where
grd1 : $m \notin deliveredMsgs$
grd2 : $m \in Msgs$
then
actDeliv : $deliveredMsgs :\in \{deliveredMsgs, (deliveredMsgs \cup \{m\})\}$

Figure 3. Machine Model 0

4.2.2. Model M1

In the previous model, it was not possible to guarantee that all messages would eventually be delivered. Actually, it was possible that messages get lost (or that the event deliver do nothing). Now, we will add a set of messages that have been generated in the network and have not yet been delivered (called $currentMsgs$). The main invariant is that there is no message that can be currently in the network and have been delivered (**invMsgs**). We included an event that generates messages (event *SEND*, it just picks a message that has not yet been generated and includes it in the $currentMsgs$ set. Invariant, **invEnable** assures that the occurrence of event *SEND* enables the occurrence of event *RECEIVE* (it makes the guard of *RECEIVE* true). The event *RECEIVE* was then refined to actually remove the message from the $currentMsgs$ set and include it in the $deliveredMsgs$ set. The variant assures that, if we consider just the deliver event, the set of delivered messages always grows (formally, this is represent by the fact that the number of elements of set $Msgs$ minus set $deliveredMsgs$ decreases). Since the set of all messages that should be delivered is finite, this variant guarantees that eventually all messages will reach their destinations. The status $convergent$ in the description of an event means that this event should be considered to prove the variant. Since all occurrences of event *SEND* enable an occurrence of *RECEIVE*, the former can be seen as an intermediate step to deliver

messages. This model is shown in Figure 4. Note that, since this model is a refinement of M0, we only present in Figure 4 the newly introduced variables and invariants, as well as new guards and new/modified actions in the events.

MACHINE *M1*
REFINES M0
VARIABLES
$currentMsgs \subseteq Msgs$

INVARIANTS
invMsgs : $deliveredMsgs \cap currentMsgs = \varnothing$
invEnable : $(\exists m \cdot m \in currentMsgs \Rightarrow (m \in Msgs \wedge m \notin deliveredMsgs))$
VARIANT
$Msgs \setminus deliveredMsgs$

EVENTS
Event *SEND* $\widehat{=}$
any m
where
grd1 : $m \in Msgs$
grd2 : $m \notin currentMsgs$
grd3 : $m \notin deliveredMsgs$
then
actCurrent : $currentMsgs := currentMsgs \cup \{m\}$
Event *RECEIVE1* $\widehat{=}$ **refines** *RECEIVE*
status CONVERGENT
any m
where
grd3 : $m \in currentMsgs$
then
actDeliv : $deliveredMsgs := deliveredMsgs \cup \{m\}$
actCurr : $currentMsgs := currentMsgs \setminus \{m\}$

Figure 4. Machine Model M1

4.2.3. Model M1loss

Alternatively, *RECEIVE* event could be refined in two different events, one delivering a message and another losing it (removing from $currentMsgs$ and not including it in $deliveredMsgs$). In this case, it is possible to prove that eventually all messages would either be delivered or lost. Due to space limitations, this model will not be shown here.

4.2.4. Model M2

To be able to show that messages arrive in the same order in which they were generated, the first step is to assume a generation order. This will be done in this model, called M2, that refines model M1. We used an auxiliary variable called $generatedMsgs$, that is just the union of the sets of current and delivered messages. To generate messages in some order, we assume that each message in the set of messages to be delivered (set $Msgs$) has a unique message identifier (given by function $idmsg$). This identifier is a natural number, and the order in which messages are generated will obey the $less$ order between

natural numbers (that is a total order). The invariant that guarantees the generation order (**invGenOrder**) states that all messages that have been generated in the network must have identifiers that are less than identifiers of non-generated messages. This model is depicted in Figure 5. Note that, with respect to model M1, only event *SEND* is refined, and therefore the variant remains true (assuring that all messages will be eventually delivered). In the *SEND2* event we require, additionally to the guards of *SEND1*, that the message that was chosen to be generated has the least identifier among all non-generated messages.

MACHINE *M2*
REFINES M1
VARIABLES
$generatedMsgs \subseteq Msgs$

INVARIANTS
invGen : $deliveredMsgs \cup currentMsgs = generatedMsgs$
invGenOrder: $\forall mc \cdot \forall md \cdot mc \in generatedMsgs \wedge md \in Msgs \setminus (generatedMsgs) \Rightarrow idmsg(md) > idmsg(mc)$

EVENTS
Event *SEND2* $\widehat{=}$ **refines** *SEND*
any m
where
grd4: $m = next(Msgs \setminus generatedMsgs)$
then
actGen: $generatedMsgs := generatedMsgs \cup \{m\}$
Event *RECEIVE2* $\widehat{=}$ *RECEIVE1*

Figure 5. Machine Model M2

4.2.5. Model M2ordered

To be able to prove that messages are delivered in the same order in which they were generated, we need, additionally to the invariant **invGenOrder** of the previous model, to guarantee that any message that have been delivered has an identifier that is less than all identifiers of messages currently in the network. This is stated by invariant **invOrder**, that is satisfied by model M2ordered, that is a refinement of model M2.

MACHINE *M2ordered*
REFINES M2
INVARIANTS
invOrder : $\forall mc \cdot \forall md \cdot mc \in currentMsgs \wedge md \in deliveredMsgs \Rightarrow idmsg(mc) > idmsg(md)$

EVENTS
Event *SEND2order* $\widehat{=}$ *SEND2*
Event *RECEIVE2order* $\widehat{=}$ **refines** *RECEIVE2*
status CONVERGENT
any m
where
grd4 : $m = next(currentMsgs)$

Figure 6. Machine Model M2ordered

4.3. Group Communication

To model group communication, we use both the general as well as the group data structures (Figures 1 and 2). Now, each message in the $Msgs$ set has the information about its source process and its target group. Therefore, when one message from this set is chosen to be delivered, one copy is generated for each of the processes belonging to the target group, building the $currentGMsgs$ set. When messages are delivered to one of the processes of the group, they are transferred to the $deliveredGMsgs$ set. Only when the last copy of the same message is delivered, the message is transferred from the $currentMsgs$ to the $deliveredMsgs$ (refining the event *RECEIVE* from the previous models). The invariants that describe the different group communication models are defined over the $currentGMsgs$ and $deliveredGMsgs$ sets. In the following we present the basic model for group communication (model M3), and then refine it to generate 3 different models, corresponding to FIFO order, causal order and total order of message delivery.

4.3.1. Model M3

As discussed above, the variables introduced in this model are $currentGMsgs$ and $deliveredGMsgs$ sets, that are similar to their counterparts in the point-to-point communication, but have several copies of the same message, one for each of its target processes. Instead of just messages, these sets contain pairs of message and target process. This model is constructed as a refinement of model M2, assuring that all messages are eventually delivered. A number of invariants were included that guarantee the consistency between the existing and the new variables. For example, invariant **invCons1** assures that only copies of messages that are in the $currentMsgs$ set can be in the $currenGMsgs$ set. Invariant **invCons2** guarantees that for any message m that is in $currentMsgs$, all copies of m (described by restricting the set $currentGMsgs$ to the domain $\{m\}$) are actually addressed to targets of m. Analogous invariants exist for delivered messages. Finally, invariant **invCon6** states that for any message m that was generated in the network, there is one copy for each of its target processes (this copy can be either in the $currentGMsgs$ set or in the $deliveredGMsgs$ set). Moreover, there can be no copy of this message to any process that does not belong to the members of the target group. The variant assures that the set of copies of messages that should be delivered is always becoming smaller (and thus eventually all messages will be delivered). There is still an event that describes the complete delivery of one message (the delivery of the last of the copies), and this event refines the one of the more abstract level. We include a new event that model the delivery of messages that are not the last one. The guards are practically the same, except of guard **grd7**, that checks whether the copy is the last one or not. The behavior of the event, considering the sets $currentGMsgs$ and $deliveredGMsgs$ is the same, but the partial receive event does not update other variables (and therefore, is not observable at the abstract level). The next three models are refinements of this model.

4.3.2. Model M3fifo

To model FIFO communication, we defined invariant **invFIFO**. This invariant states that whenever there are two different messages mc and md having the same process p as

MACHINE *M3*
REFINES M2
VARIABLES
$currentGMsgs \subseteq (currentMsgs \times Procs)$

$deliveredGMsgs \subseteq (generatedMsgs \times Procs)$

INVARIANTS
invDelCurr : $deliveredGMsgs \cap currentGMsgs = \varnothing$
invCons1 : $currentMsgs = dom(currentGMsgs)$
invCons2 : $\forall m \cdot m \in currentMsgs \Rightarrow \{m\} \lhd currentGMsgs \subseteq (\{m\} \times members(target(m)))$
invCons3 : $deliveredMsgs \subseteq dom(deliveredGMsgs)$
invCons4 : $\forall m \cdot m \in deliveredMsgs \Rightarrow m \notin dom(currentGMsgs)$
invCons5 : $\forall m \cdot m \in deliveredMsgs \Rightarrow \{m\} \times members(target(m)) \subseteq deliveredGMsgs$
invCons6 : $\forall m \cdot m \in generatedMsgs \Rightarrow (\{m\} \lhd currentGMsgs) \cup (\{m\} \lhd deliveredGMsgs) = (\{m\} \times members(target(m)))$

EVENTS
Event *SEND3* $\widehat{=}$ **refines** *SEND2*
any m
then
actCurrentG : $currentGMsgs := currentGMsgs \cup (\{m\} \times members(target(m)))$
Event *RECEIVE3* $\widehat{=}$ **refines** *RECEIVE2*
status CONVERGENT
any m, p
where
grd4 : $p \in Procs$
grd5 : $p \in members(target(msg))$
grd6 : $p \notin ran(\{m\} \lhd deliveredGMsgs)$
grd7 : $ran(\{m\} \lhd deliveredGMsgs) \cup \{p\} = members(target(msg))$
then
actCurrentG : $currentGMsgs := currentGMsgs \setminus \{(m \mapsto p)\}$
actDeliverG : $deliveredGMsgs := deliveredGMsgs \cup \{(m \mapsto p)\}$
Event *PART-RECEIVE3* $\widehat{=}$
status CONVERGENT
any m, p
where
grd1 : $msg \notin deliveredMsgs$
grd2 : $msg \in Msgs$
grd3 : $msg \in currentMsgs$
grd4 : $p \in Procs$
grd5 : $p \in members(target(msg))$
grd6 : $p \notin ran(\{m\} \lhd deliveredGMsgs)$
grd7 : $ran(\{m\} \lhd deliveredGMsgs) \cup \{p\} \subset members(target(msg))$
then
actCurrentG : $currentGMsgs := currentGMsgs \setminus \{(m \mapsto p)\}$
actDeliverG : $deliveredGMsgs := deliveredGMsgs \cup \{(m \mapsto p)\}$

Figure 7. Machine Model M3

target, one of them currently in the network (mc) and the other already delivered (md), and the source of these two messages is the same process, the identifier of message mc must be greater than the one of md (remind that messages with smaller identifier are always generated before messages with greater ones). This assures that a process can not receive messages from another process in an order that was different from the one they were generated. Note that this invariant does not restrict the order in which messages from different processes are received. To implement this model, we just need to include one extra guard in the receipt events (guard **grd8**), that assures that the message being chosen to be delivered (m) has the least identifier among all messages from the same source process that are currently in the network. Due to lack of space, we will only depict event *RECEIVE3fifo*, event *PART-RECEIVE3fifo* is analogous.

MACHINE *M3fifo*
REFINES M3
INVARIANTS
invFIFO: $\forall p \cdot \forall mc \cdot \forall md \cdot p \in Procs \land (mc \mapsto p) \in currentGMsgs \land (md \mapsto p) \in deliveredGMsgs \land source(mc) = source(md) \Rightarrow idmsg(mc) > idmsg(md)$

EVENTS
Event *SEND3fifo* $\widehat{=}$ *SEND3*

Event *RECEIVE3fifo* $\widehat{=}$ **refines** *RECEIVE3*
status CONVERGENT
any m
where
grd8: $\forall msg \cdot msg \in currentMsgs \land source(m) = source(msg) \Rightarrow idmsg(m) < idmsg(msg)$

Figure 8. Machine Model M3fifo

4.3.3. Model M3total

Total order is achieved by invariant **invTOTAL** requiring that if a message $m1$ has been delivered for process $p1$ and there is another message $m2$ for $p1$ currently in the network, if $m2$ has already been delivered for any other process $p2$ that is also in the target group of message $p1$, than $m1$ must have also been delivered for $p2$. This assures that messages to the same targets arrive in the same order. The implementation of this behavior occurs in the receive events, that must have an additional guard (**grd8**) guaranteeing that, whenever message m is chosen to be delivered to process p, p has already received all messages that all other processes that have also m as target have received.

4.3.4. Model M3causal

Finally, to model causal order we included a new variable called $order$, that records the causal order in which messages are delivered. This order is built incrementally during execution: once a message is generated, it causally depends on all messages that the source process of this message have received. This is described by action **actOrder** in the event *SEND3causal*. The invariant that describes causal order is **invCAUSAL**, that states that if a process p received a message $m1$ and did not receive $m2$ yet, than $m2$ can not be a

MACHINE *M3total*
REFINES M3
INVARIANTS
invTOTAL: $\forall p1 \cdot \forall p2 \cdot \forall m1 \cdot \forall m2 \cdot$
$((m1 \mapsto p1) \in deliveredGMsgs \wedge (m2 \mapsto p1) \in currentGMsgs) \Rightarrow$
$((m2 \mapsto p2) \in deliveredGMsgs \Rightarrow (m1 \mapsto p2) \in deliveredGMsgs)$

EVENTS
Event *SEND3total* $\widehat{=}$ *SEND3*

Event *RECEIVE3total* $\widehat{=}$ **refines** *RECEIVE3*
status CONVERGENT
any m
where
grd8: $\forall p1 \cdot p1 \in members(target(msg)) \Rightarrow (\exists m1 \cdot (m1 \mapsto p1) \in deliveredGMsgs \wedge p \in members(target(m1)) \Rightarrow (m1 \mapsto p) \in deliveredGMsgs)$

Figure 9. Machine Model M3total

cause of $m1$. Upon reception of a message, we have to make sure that all possible causes of this message have already been received (guard **grd8**). In Figure 10 we omitted some invariants about the causal order (assuring that is is irreflexive, antisymmetric, transitive, and that the causal order is compatible with the order of identifiers).

MACHINE *M3causal*
REFINES M3
VARIABLES
$order \subseteq generatedMsgs \times generatedMsgs$

INVARIANTS
invCAUSAL: $\forall p \cdot \forall m1 \cdot \forall m2 \cdot ((m1 \mapsto p) \in deliveredGMsgs \wedge (m2 \mapsto p) \in currentGMsgs) \Rightarrow (m2 \mapsto m1 \notin order)$

EVENTS
Event *SEND3causal* $\widehat{=}$ **refines** *SEND3*
any m
then
actOrder : $order := order \cup ((dom(deliveredGMsgs \rhd \{source(m)\}) \cap dom(target \rhd \{target(m)\})) \times \{m\})$
Event *RECEIVE3total* $\widehat{=}$ **refines** *RECEIVE3*
status CONVERGENT
any m, p
where
grd8: $\forall msg \cdot msg \mapsto p \in currentGMsgs \wedge m \neq msg \Rightarrow msg \mapsto m \notin order$

Figure 10. Machine Model M3causal

5. Using the Library of Communication Models

With the proposed library of communication models, a developer of a distributed algorithm would first choose the appropriate communication model specification, matching the assumptions of the algorithm in terms of communication type (group or point-to-point) and message ordering, and then would refine that specification to represent the specific aspects of the distributed algorithm.

As an example, we have modeled a mutual exclusion distributed algorithm based on the one proposed by [Ricart and Agrawala 1981]. That algorithm defines how a group of processes should collaborate to assure deadlock- and starvation-free mutual exclusion. A process that wants to enter the critical section sends a request to all group members. Each member has to eventually answer the request giving permission to the originator. When the originator receives permissions from all the processes, then it is allowed to enter the critical section. The access is assured to be linearized because a process in the critical section or processes that compete for the critical section, and have priority over others, do not send back permissions until they leave their critical session. Priority is decided homogeneously using a message sequence number scheme.

In our exercise, instead of dealing with sequence numbers, we choose the group communication system to offer the total order delivery and build a mutual exclusion protocol based on that. To entry the critical session the procedure is the same as above. Now, relying on the total ordering of messages, when the originator process receives its own request it delays answers to requests by others until it leaves its own critical section. Any request by other processes that arrives prior to its own request is answered with permission. This example is interesting because the fundamental properties of the algorithm are built relying on the communication semantics which is explicitly stated and proved in the library. Due to space restrictions a detailed presentation of the example is not possible.

6. Final Considerations

As mentioned before, formal specification and analysis of distributed algorithms is not recent. Most commonly, one can observe the use of specification languages such as Pi-calculus [Milner 1999], CSP [Hoare 1985], I/O Automata [Lynch and Tuttle 1989], PROMELA [Holzmann 1997], that support notions of processes and communication channels. PROMELA, for instance, supports channels suited to model one to one communication. I/O Automata and some process calculi languages allow the synchronization of multiple processes in a channel. While this can be used as an abstraction to group communication, it does not represent important details in an asynchronous distributed setting where arbitrary communication delays may happen between any two pairs of processes and give rise to several possible delivery orders.

Efforts for a formal representation of middleware and some available communication mechanisms can be witnessed in the literature. For instance, in [Bryans et al. 2009] the authors model middleware behavior to assess its impact in the design of distributed applications. As a part of that, some aspects of the communication platform are modeled. However, only a restricted subset of communication models is considered. In [Hoang et al. 2009] the authors propose the use of specification patterns as means to foster the reuse of specifications and proofs. As example, they model some one-to-one communication mechanisms as patterns. A system designer could then adopt one such pattern in a guided refinement step, what would ease the integration of communication mechanisms to the design of a system. Despite these contributions, we could not find support for a more comprehensive set of communication models in a way that would enable reuse of specifications and proofs when designing and reasoning about distributed algorithms.

Differently of building a specific distributed algorithm, in this paper we have proposed a library of specifications, for several communication semantics, with their main

properties proven. Approximately 170 proof obligations were generated by the platform and proved interactively. A developer of a distributed algorithm can reuse the specification that matches the assumptions of the algorithm in terms of type, ordering and reliability of the communication mechanism. This reuse includes the reuse of all existing proofs.

This was the first step towards the construction of a formal framework for the verification of distributed algorithms using theorem proving. Ongoing work is the modeling and verification of distributed algorithms as case studies. Future work includes extending modeling abstractions to other features of group communication.

References

Abrial, J. R. (2005). *The B-Book: Assigning Programs to Meanings*. Cambridge University Press.

Abrial, J.-R. (2010). *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition.

Birman, K. P. (1996). *Building secure and reliable network applications*. Manning Publications, USA.

Bryans, J., Fitzgerald, J., Romanovsky, A., and Roth, A. (2009). Formal modelling and analysis of business information applications with fault tolerant middleware. In *Proceedings of the 2009 14th IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS '09, pages 68–77. IEEE Computer Society.

DEPLOY Project. Event-b and the rodin platform. http://www.event-b.org/ (last accessed April 2nd 2012).

Hoang, T. S., Fürst, A., and Abrial, J.-R. (2009). Event-B patterns and their tool support. In Hung, D. V. and Krishnan, P., editors, *SEFM*, pages 210–219. IEEE Computer Society.

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice Hall.

Holzmann, G. J. (1997). The model checker SPIN. *IEEE Trans. on Soft. Eng.*, 23(5):279–295.

Lamport, L. (1978). Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565.

Laprie, J.-C., Randell, B., Avizienis, A., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33.

Lynch, N. and Tuttle, M. (1989). An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246.

Milner, R. (1999). *Communicating and mobile systems: the π-calculus*. Cambridge University Press, USA.

Ribeiro, L., dos Santos, O. M., Dotti, F. L., and Foss, L. (2011). Correct transformation: From object-based graph grammars to promela. *Science of Computer Programming*, In Press, Corrected Proof:–.

Ricart, G. and Agrawala, A. K. (1981). An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, 24(1):9–17.

Um Verificador de Modelos Descritos em Redes de Autômatos Estocásticos*

Claiton M. Correa, Fernando L. Dotti, Paulo Fernandes, Eli Maruani, Lucas G. Oleksinski, Afonso Sales

[1]Faculdade de Informática – Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Programa de Pós-Graduação em Ciência da Computação
Av. Ipiranga, 6681 Prédio 32 – 90.619-900 – Porto Alegre – RS – Brasil

{claiton.correa, eli.maruani, lucas.oleksinski}@acad.pucrs.br,
{fernando.dotti, paulo.fernandes, afonso.sales}@pucrs.br

Abstract. *During the construction of modern systems, both the satisfaction of functional properties and the ability to assert levels of performance and availability of these systems are necessary. In this sense, several modelling formalisms rely on abstractions and tools to support both the verification of functionality and perform quantitative analysis. Stochastic Automata Networks is a formalism focused on the evaluation of performance, which is attractive both because of its set of abstractions and the potential treatment of large state space. This article describes the first version of a model checker for Stochastic Automata Networks. This model checker uses Symbolic Model Checking techniques and verifies properties written in a temporal logic called Computation Tree Logic. Its use with a classic case study is reported.*

Resumo. *Durante a construção de sistemas modernos, tanto a satisfação de propriedades funcionais quanto a habilidade de afirmar níveis de desempenho e disponibilidade destes sistemas são necessárias. Neste sentido, vários formalismos de modelagem contam com abstrações e ferramentas para apoiar tanto a verificação da funcionalidade como realizar análises quantitativas. Redes de Autômatos Estocásticos são um formalismo voltado para a avaliação de desempenho, o qual é atrativo tanto devido a seu conjunto de abstrações como pelo potencial tratamento de amplo espaço de estados. Este artigo descreve a primeira versão de um verificador de modelos descritos em Redes de Autômatos Estocásticos. Este verificador faz uso de técnicas de verificação simbólica e verifica propriedades escritas na lógica temporal Computation Tree Logic. Relato do seu uso em estudos de casos clássicos são feitos.*

1. Introdução

Sistemas computacionais são utilizados em aplicações que controlam diferentes tipos de serviços, dos mais simples aos mais complexos e críticos. A crescente demanda por sistemas com estas características fez com que importantes técnicas de validação fossem desenvolvidas para verificar corretude e confiabilidade, visto que consequências nocivas podem acontecer a partir de um mau funcionamento de um *software*. Porém, garantir

*Projeto com suporte financeiro FAPERGS e CNPq.

que um sistema ou modelo esteja correto, englobando todas as características inicialmente definidas, não é uma tarefa fácil. Nos projetos de *hardware* e sistemas complexos mais tempo e esforço são gastos na verificação do que na construção dos mesmos [Clarke et al. 1999].

A abordagem de verificação de modelos conhecida como *Model Checking* é uma técnica para verificação de sistemas concorrentes de estados finitos. Apesar desta restrição, a técnica apresenta o benefício de que toda a verificação pode ser realizada automaticamente. *Model Checking* é aplicável a muitas classes importantes de sistemas e áreas do conhecimento. Dentre elas, pode-se citar o uso de verificação para controladores de *hardware*, *softwares* críticos, busca de erros em estruturas de dados, dentre outros. Ainda, possui grandes vantagens sobre abordagens tradicionais para este problema, as quais são baseadas em simulação, testes e raciocínio dedutivo [Clarke et al. 1999].

Complementarmente, formalismos e técnicas para a avaliação de desempenho de sistemas são importantes para atestar, tão logo possível durante o projeto de tais sistemas, que os mesmos atendem aos requisitos, tipicamente temporais, mas também espaciais, inseridos aos sistemas. Redes de Autômatos Estocásticos (SAN - *Stochastic Automata Networks*) são um formalismo voltado à avaliação de desempenho que permite modelagem de sistemas com grande espaço de estados [Plateau 1985]. O formalismo SAN busca prover uma forma compacta de descrever realidades complexas, modelando sistemas como um conjunto de subsistemas quase independentes que interagem ocasionalmente. Cada subsistema é composto de autômatos, onde cada autômato possui estados e transições entre eles, que podem ser realizadas através de eventos locais e/ou sincronizados. SAN permite também a representação de dependências funcionais, onde o comportamento de um autômato pode ser dependente do estado global em que a rede se encontra.

A definição de propriedades a serem verificadas em um modelo pode ser feita utilizando-se lógicas temporais, as quais descrevem a ordem em que eventos devem ocorrer no tempo sem a introdução de tempo de maneira explícita [Baier and Katoen 2008]. A introdução de lógicas temporais para *Model Checking* veio em meados dos anos 1980 por Clarke e Emerson, permitindo que este tipo de técnica de validação seja feita de forma automatizada e eficiente [Clarke et al. 1999].

A possibilidade de verificação de propriedades funcionais de modelos SAN agrega valor aos esforços de modelagem usando SAN, possibilitando um leque maior de análises a partir de um mesmo modelo. De fato, diversos formalismos para avaliação de desempenho contam com possibilidade de verificação, tal como Redes de Petri, Álgebra de Processos e Cadeias de Markov.

Este artigo descreve a primeira versão de um verificador de modelos para SAN. A ferramenta verifica se uma propriedade descrita em *Computation Tree Logic* é satisfeita dado um modelo SAN de entrada. Este artigo descreve sucintamente Redes de Autômatos Estocásticos (Seção 2); Verificação de Modelos (Seção 3); introduz a Lógica Temporal CTL (Seção 3.1); Diagramas de Decisão Multi-Valorada (Seção 4); e por fim descreve a arquitetura da ferramenta (Seção 5), bem como, conclui o trabalho (Seção 6).

2. Redes de Autômatos Estocásticos

Redes de Autômatos Estocásticos são um formalismo para modelagem de sistemas com grandes espaços de estados [Plateau 1985], e possuem equivalência de descrição com

Cadeias de Markov [Stewart 2009]. A idéia central por trás deste formalismo é prover a modularidade, permitindo que se modelem sistemas divididos em vários subsistemas de forma quase independente e ocasional interação [Benoit et al. 2003].

Um modelo SAN é descrito por um conjunto de autômatos que são compostos por estados e suas transições. Essas transições possuem uma lista de eventos associados que podem ser locais ou sincronizantes. Eventos locais alteram o estado de um único autômato, sem interferir no comportamento dos demais enquanto eventos sincronizantes estão associados a transições de dois ou mais autômatos simultaneamente, alterando o estado local de todos os envolvidos [Fernandes et al. 1998]. Como exemplo, tem-se a rede da Figura 1, problema do jantar dos filósofos, composta por três filósofos, neste caso com apenas transições locais.

Cada evento possui uma taxa de transição, que pode ser constante ou funcional, e caso um mesmo evento possa transitar para mais de um estado, uma probabilidade é associada para cada transição possível, podendo também ser constante ou funcional. Taxas e probabilidades funcionais são funções associadas ao evento, onde a função é avaliada sobre o estado global da rede e retorna um valor numérico que decidirá sobre o disparo ou não da transição [Brenner et al. 2005]. Esta é uma das vantagens na utilização de SAN em relação a outros formalismos. Pode-se consultar, por exemplo, o estado de um autômato através da operação *st* da seguinte forma: (st Autômato == Estado). Outra função comum é *nb*, que retorna a quantidade de autômatos que se encontram em um determinado estado (nb Estado).

Através de funções, pode-se construir dependências entre autômatos já que uma transição em um autômato pode estar habilitada ou não dependendo da função associada a sua taxa, a qual leva em consideração estados de outros autômatos. A sintaxe e outras funções SAN se encontram em [Benoit et al. 2007]. Na Figura 1, o evento F4 apresenta um exemplo de taxa funcional, onde a transição somente ocorrerá caso o estado do autômato Phil3 for diferente de “Right”.

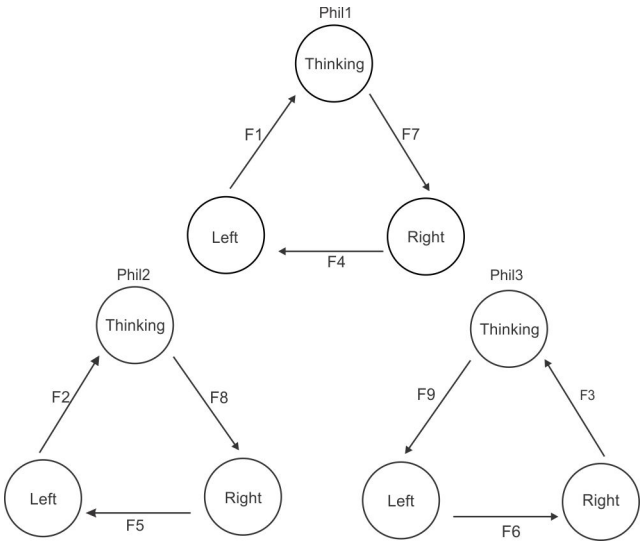
Em um modelo SAN é necessário definir uma função de atingibilidade descrevendo no modelo um ou mais estados que são conhecidamente atingíveis, pois através destes será calculado o espaço de estados atingível.

Além do uso de funções, a simplicidade e similaridade com Cadeias de Markov [Brenner et al. 2007] e o uso da álgebra tensorial generalizada também são vantagens ao se trabalhar com Redes de Autômatos Estocásticos.

Para compreensão da Figura 1, é importante conhecer o significado dos estados no modelo. Quando um filósofo quer comer, pega o garfo que está a sua direita, caso destro, ou à esquerda, caso canhoto, caracterizando a transição do estado “pensando” para o estado “quer comer”. Ao pegar o segundo garfo passa para o estado “comendo”. Nesse exemplo, modelo com três filósofos, apenas o último filósofo, filósofo 2, é canhoto.

3. Verificação de Modelos (*Model Checking*)

Model Checking pode ser definida como uma técnica automática que objetiva verificar se uma propriedade é satisfeita ou não por um modelo. Para esta abordagem, deve-se abstrair um sistema, ou seja, modela-lo em um sistema de transições de estados finitos, e descrever propriedades a serem verificadas em lógicas temporais, abordadas na Seção



TIPO	EVENTO	TAXA
LOCAL	F1	α
LOCAL	F2	α
LOCAL	F3	α
LOCAL	F4	(ST Phil3 != Right) * λ
LOCAL	F5	(ST Phil1 != Right) * λ
LOCAL	F6	(ST Phil1 != Left) * λ
LOCAL	F7	(ST Phil2 != Left) * λ
LOCAL	F8	(ST Phil3 != Left) * λ
LOCAL	F9	(ST Phil2 != Right) * λ

partial reachability = ((nb Thinking) == 3)

Figura 1. SAN que representa o problema do jantar dos filósofos para três filósofos.

3.1.

De acordo com [Clarke et al. 1999], o problema de *Model Checking* é definido como dado um modelo M de um sistema descrito em um sistema de transição de estados finitos TS e uma fórmula f, verifica-se se há estados s tal que, $s \models f$.

Essa verificação é feita de forma sistemática e como resultado retorna para o usuário uma testemunha, caso a propriedade seja satisfeita, ou um contra-exemplo caso não satisfeita. Uma testemunha indica um caminho de computação onde pode se verificar a veracidade da propriedade, enquanto um contra-exemplo apresenta uma sequência de estados que indica a não satisfação da fórmula. A possibilidade de geração de contra-exemplos e testemunhas é uma das principais vantagens em relação a outras técnicas de validação, pois seu estudo fornece um conjunto muito importante de informações para depuração do sistema sob verificação.

3.1. Lógicas Temporais

Lógica temporal é um formalismo para descrever sequências de transições entre estados em um sistema reativo. Neste tipo de lógica, o tempo não é mencionado explicitamente. Ao invés disto, uma fórmula da lógica temporal define uma relação de ordem entre estados que obedecem certas propriedades [Clarke et al. 1999]. Dentre os tipos de lógica podemos citar a *Linear Temporal Logic* (LTL), que trabalha sob uma perspectiva de tempo linear e a *Computation Tree Logic* (CTL) que trabalha sob uma perspectiva de tempo ramificado, ou seja, sob uma árvore de estados ramificada, sendo esta última abordada neste trabalho.

3.1.1. Computation Tree Logic (CTL)

Computation Tree Logic (CTL) é uma lógica temporal baseada na lógica proposicional com noção discreta de tempo. O objetivo da mesma é ser uma lógica de grande expressividade para a formulação de um importante conjunto de propriedades de sistemas. É

definida em termos de uma árvore infinita de estados onde o futuro não é determinado [Baier and Katoen 2008].

As fórmulas CTL permitem que se expresse propriedades de algumas ou todas as computações que iniciem em um determinado estado. As fórmulas são compostas de no mínimo três partes: quantificadores de caminho, que podem ser o operador universal $\forall$ (pronunciado "para todos os caminhos") ou o operador existencial $\exists$ (pronunciado "existe um caminho"), operadores temporais, que são ($\bigcirc$, **U**, $\Box$ e $\Diamond$) e/ou conectores booleanos e proposições atômicas, que são assertivas descritas sobre um modelo.

Os operadores temporais da CTL são definidos da seguinte maneira:

- $\bigcirc$ - simbolizando "no próximo estado", também representado por **X**.
- $\Diamond$ - simbolizando "em um estado futuro", também representado por **F**.
- $\Box$ - simbolizando "globalmente", também representado por **G**.
- **U** - simbolizando "até que".

Uma fórmula CTL possui sintaxe definida pela seguinte gramática:

$$\Phi ::= \text{true} \mid a \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \exists\varphi \mid \forall\varphi$$

$$\varphi ::= \bigcirc\Phi \mid \Phi_1 U \Phi_2$$

Maiores detalhes sobre a semântica CTL são encontrados em [Hurth and Ryan 2004].

Formas normais para CTL foram criadas para se reduzir o escopo de operações a serem tratadas, o que torna menos complexa a implementação de um algoritmo de verificação, por exemplo. Através de regras de equivalência, toda fórmula CTL pode ser convertida para uma forma normal. A Forma Normal Existencial (*ENF - Existential Normal Form*) tem como característica trabalhar apenas com o quantificador de caminho existencial $\exists$, onde os operadores básicos $\exists\Box$, $\exists$U, e $\exists\bigcirc$ são suficientes para definir toda a sintaxe da CTL. Detalhes sobre a Forma Normal Existencial são encontrados em [Baier and Katoen 2008].

4. Multivalued Decision Diagrams (MDD)

Diagramas de decisão são grafos acíclicos dirigidos comumente usados para codificar de forma eficiente funções booleanas, conjuntos de estados e transições de formalismos estocásticos, ou ainda uma gama de outras funções arbitrárias [Clarke et al. 1999]. Sua aplicação está em diversos trabalhos, como verificação de circuitos aritméticos e representação de Cadeias de Markov de tempo discreto e contínuo.

Multivalued Decision Diagrams (MDD) ou Diagramas de Decisão Multi-Valorada aceitam múltiplos valores nas arestas de ligação entre os nodos e possuem as seguintes propriedades:

- Nodos são organizados em $N + 1$ níveis, onde N é o número de componentes do sistema;
- Nível N possui apenas um nodo não terminal r (nodo *root*), enquanto do nível $N - 1$ até o nível 1 há um ou mais nodos não terminais;
- Nível 0 tem dois nodos terminais: **0** e **1**;
- Um nodo não terminal p em um nível l contêm n_l arcos apontando para nodos no nível l - 1.
- Não existem nodos duplicados.

5. *Stochastic Automata Networks Model Checker*

Esta Seção descreve uma ferramenta que realiza CTL *Model Checking* para modelos descritos em SAN. Sua implementação foi feita usando linguagem de programação C++ e se baseia no algoritmo de satisfação de fórmulas CTL descrita em [Baier and Katoen 2008] (página 348). A Figura 2 apresenta um detalhamento dos processos envolvidos no verificador onde retângulos são processos e paralelogramos são dados.

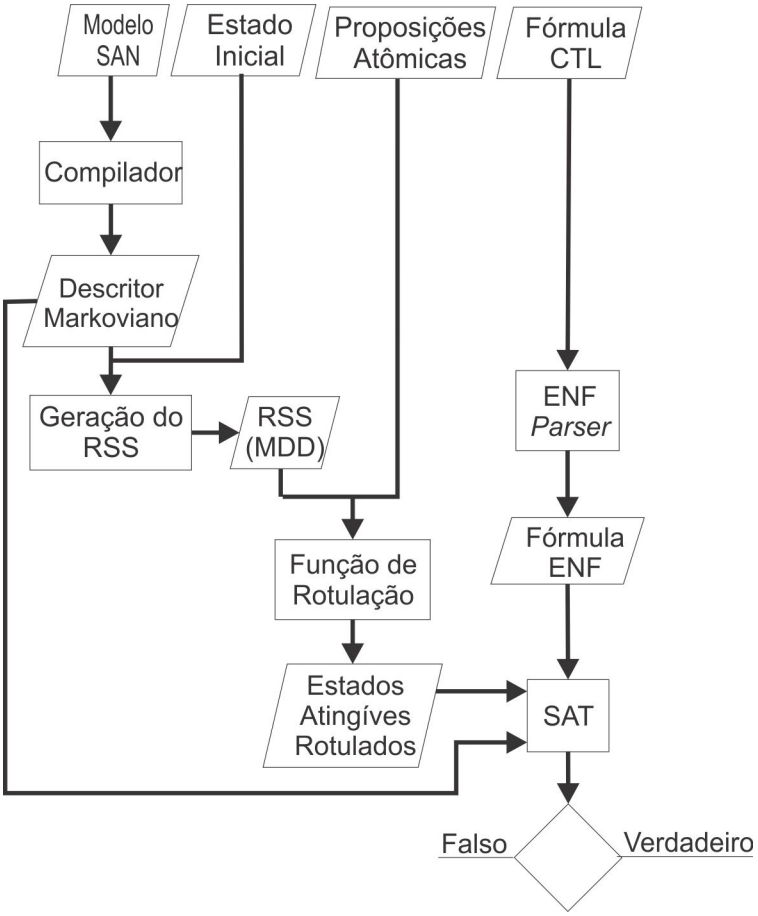


Figura 2. Arquitetura do verificador

- Modelo SAN - O modelo SAN de entrada deve ser descrito conforme abordado na Seção 2. Na ferramenta, um modelo SAN é descrito em formato textual que deve seguir as regras sintáticas descritas em [Benoit et al. 2007], tendo em vista que o compilador utilizado é o mesmo do software PEPS - *Performance Evaluation of Parallel Systems*[1].
- Estado Inicial - Um ou mais estados iniciais conhecidamente atingíveis devem ser definidos na função de atingibilidade do modelo SAN de entrada, pois esta função será o ponto de partida para a geração do RSS.
- Proposições atômicas - Proposições atômicas são expressões SAN que, assim como funções (Seção 2) podem ser avaliadas sobre o estado global da rede, retornando verdadeiro ou falso para cada estado. Elas denotam afirmações sobre a

[1] Acesso em: http://www-id.imag.fr/Logiciels/peps/

rede e são definidas no modelo SAN de entrada conforme exemplos apresentados na Seção 5.1.

- Fórmula CTL - Através de fórmulas CTL para expressões SAN são descritas propriedades sobre o modelo de entrada, conforme exemplos apresentados na Seção 5.1.
- Compilador - Após a entrada dos dados, a compilação do modelo SAN é iniciada, gerando o Descritor Markoviano que tem o objetivo de representar implicitamente as transições da Cadeia de Markov subjacente. Este descritor é que um conjunto de tensores que, ao serem operados por álgebra tensorial, permitem que se alcance todos os estados da Cadeia de Markov de maneira implícita, sendo assim extremamente eficiente para manipulação e armazenamento. As operações que podem ser efetuadas (soma tensorial ou produto tensorial) reproduzem as interações e como os diferentes autômatos sincronizam o disparo de eventos [Czekster 2010].
- Geração do RSS - O Descritor Markoviano gerado pela compilação é utilizado como Função de Transição do sistema. Logo, dado um estado qualquer do modelo, é verificado nos tensores do Descritor se há eventos locais ou sincronizados que podem ser disparados a partir deste dado estado. Se o evento tiver uma taxa constante, logo ele pode ser disparado e então o estado que é atingível a partir do dado estado é contabilizado (no caso, este estado atingível é incluído no RSS que está sendo calculado). Como modelos Markovianos estruturados possuem o conhecido problema da explosão do espaço de estados, esse processo se torna oneroso pois conforme cresce o número de autômatos e/ou estados desses autômatos, cresce exponencialmente o espaço de estados gerado, tornando problemático também seu armazenamento. Para armazenar o RSS de forma mais compacta, foi utilizado MDDs para codificar o espaço de estados, pois oferecem uma solução melhor manipulável e de armazenamento econômico. Para otimizar o processo de geração do RSS a técnica de saturação foi empregada. Essa técnica dispara exaustivamente, em um nodo de um MDD em um nível k, todos os eventos ativos que afetam somente esse nível k e abaixo, até que o conjunto de estados codificado pelo nodo $\langle k.p \rangle$, onde p é o índice do nodo no nível k, alcance um ponto fixo. A estratégia é aplicada partindo do nível mais inferior, subindo em direção ao nodo *root*. Assim, cada nodo é saturado somente uma vez após todos os nodos abaixo terem sido saturados. Como resultado desse processo temos um MDD que codifica todos os estados atingíveis do modelo SAN de entrada. Mais detalhes da geração do RSS se encontram em [Sales and Plateau 2009].
- Função de Rotulação - Esta etapa tem o objetivo de rotular todos os estados com as proposições atômicas que os satisfazem. Dados o conjunto de proposições atômicas e o RSS, a função cria de forma iterativa para cada proposição atômica do modelo, um MDD que codifica os estados do RSS onde a proposição é verdadeira.
- ENF *Parser* - Outro componente da arquitetura do verificador é o ENF *Parser*. Seu objetivo é, dada uma fórmula CTL de entrada, através de regras de equivalência [Baier and Katoen 2008], convertê-la em uma formula ENF. Sua implementação percorre em pré-ordem a árvore sintática que representa a fórmula e, não estando em Forma Normal Existencial, cria-se uma nova árvore sintática substituindo as operações fora deste escopo por outra sequência de operações equivalentes que respeita a sintaxe ENF.
- Algoritmo de satisfação (SAT) - Este algoritmo é responsável pelo processo de

verificação da fórmula de entrada sobre o modelo. Sua implementação se baseou no algoritmo de satisfação de formulas CTL em Forma Normal Existencial descrito em [Baier and Katoen 2008]. O algoritmo avalia se a fórmula de entrada é satisfeita ou não pelo modelo resultando verdadeiro ou falso. Os detalhes do algoritmo e sua implementação estão descritos na Seção 5.2.

5.1. Propriedades CTL para SAN

Definir propriedades CTL para SAN exige o esforço de expressar o que se quer saber sobre um modelo através de ambas as sintaxes. Nesse caso, descrevemos propriedades através de fórmulas CTL com proposições atômicas expressas em SAN. Proposições atômicas são asserções sobre um modelo que avaliam verdadeiro/falso para cada estado global e/ou estados locais de um ou mais autômatos. Dessa forma, tendo-se a definição de SAN em [Benoit et al. 2007] e a sintaxe da CTL apresentada na Seção 3.1.1, define-se o conjunto de propriedades CTL para Redes de Autômatos Estocásticos.

Como exemplo, podemos definir as proposições atômicas abaixo:

Phi0NaoCome = (st P0 != Left);

GarfoPhi0Phi2 = ((st P0 == Left) && (st P2 == Right));

GarfoPhi2Phi1 = ((st P2 == Left) && (st P1 == Right));

GarfoPhi1Phi0 = ((st P1 == Left) && (st P0 == Right));

NinguemCome = ((st P0 != Left) && (st P1 != Left) && (st P2 != Right));

Com as proposições atômicas acima, definidas para o modelo SAN do Jantar dos Filósofos, pode-se criar fórmulas CTL a serem verificadas, como as exemplificadas abaixo:

- $\exists\Box$(Phi0NaoCome) - Existe um caminho possível onde em todos os estados o filósofo 0 não come. Essa propriedade verifica se há a possibilidade do filósofo 0 jamais atingir o estado "comendo". Teste de Postergação Indefinida.
- $\exists\Diamond$(GarfoPhi0Phi2 || GarfoPhi2Phi1 || GarfoPhi1Phi0) - Existe um caminho possível onde eventualmente filósofos vizinhos utilizam o mesmo garfo. As proposições atômicas usadas nesta propriedade testam todas as possíveis combinações de filósofos vizinhos utilizando o mesmo garfo. Sendo essa propriedade falsa temos um modelo mutuamente exclusivo. Teste de Exclusão Mútua.
- $\exists\Box$(NinguemCome) - Existe um caminho onde em todos os estados nenhum filósofo come. Esta propriedade verifica a possibilidade de haver caminhos onde nenhum filósofo atinge o estado "comendo". Sendo a fórmula satisfeita o modelo possui Deadlock. Teste de Deadlock.

Vale lembrar que a primeira propriedade, Postergação Indefinida, é um exemplo utilizando apenas o filósofo 0. Caso se deseja verificar a existência de postergação indefinida em sua completude, é necessário testar para todos os filósofos do modelo.

5.2. Algoritmo de satisfação de fórmulas CTL

Esta Seção apresenta o funcionamento do algoritmo de satisfação de fórmulas CTL em ENF proposto em [Baier and Katoen 2008] e sua implementação na ferramenta apresentada neste trabalho. Como entrada, o algoritmo precisa de um sistema de transição finito

TS, um conjunto de estados S e uma fórmula CTL escrita em ENF tendo como saída o conjunto de estados que satisfazem a fórmula. A Figura 3 apresenta o algoritmo e a Figura 4 apresenta sua implementação na ferramenta.

Input: finite transition system TS with state set S and CTL formula Φ in ENF
Output: $Sat(\Phi) = \{ s \in S \mid s \models \Phi \}$
switch(Φ):

true	:	**return** S;
a	:	**return** $\{ s \in S \mid a \in L(s) \}$;
$\Phi_1 \wedge \Phi_2$	:	**return** $Sat(\Phi_1) \cap Sat(\Phi_2)$;
$\neg\Psi$	:	**return** $S \setminus Sat(\Psi)$;
$\exists\bigcirc \Psi$	:	**return** $\{ s \in S \mid Post(s) \cap Sat(\Psi) \neq \varnothing \}$;
$\exists(\Phi_1 \,\mathsf{U}\, \Phi_2)$	:	$T := Sat(\Phi_2)$; (* compute the smallest fixed point *) **while** $\{ s \in Sat(\Phi_1) \setminus T \mid Post(s) \cap T \neq \varnothing \} \neq \varnothing$ **do** **let** $s \in \{ s \in Sat(\Phi_1) \setminus T \mid Post(s) \cap T \neq \varnothing \}$; $T := T \cup \{ s \}$; **od**; **return** T;
$\exists\Box \Phi$	:	$T := Sat(\Phi)$; (* compute the greatest fixed point *) **while** $\{ s \in T \mid Post(s) \cap T = \varnothing \} \neq \varnothing$ **do** **let** $s \in \{ s \in T \mid Post(s) \cap T = \varnothing \}$; $T := T \setminus \{ s \}$; **od**; **return** T;

end switch

Figura 3. Função Recursiva de satisfação de fórmulas CTL [Baier and Katoen 2008].

Abaixo segue de forma detalhada a explanação do algoritmo, Figura 3, associado a sua implementação, Figura 4. O algoritmo de satisfação de fórmulas CTL é recursivo e, para cada fórmula, executa as seguintes operações:

- se a fórmula for o booleano *true* retorna o espaço de estados atingível, ou seja, S. Nesse caso, ao se avaliar no código *switch*, essa operação não será encontrada no mapeamento CTL e então executará o *default*, linha 120 da Figura 4, que verificará se o conteúdo do nodo da árvore sintática em avaliação é "T", e sendo assim retorna o MDD do RSS.
- se a fórmula for uma proposição atômica "a", retorna todos os estados do RSS rotulados por "a". Na implementação, assim como no caso anterior, executará o *default*, porém não será "T", fazendo com que o MDD receba o conjunto de estados rotulados pela proposição atômica.
- se for uma conjunção ($\wedge$) realiza a intersecção das execuções de Sat(Φ_1) e Sat(Φ_2), pois ambos precisam ser verdadeiros. Na implementação equivale a entrada na linha 14 da Figura 4.
- se for uma negação ($\neg$), o algoritmo realiza a diferença entre o RSS e a execução do Sat(Ψ), retornando todos os estados do espaço de estados atingível que não satisfazem Ψ. Na implementação equivale a entrada na linha 25 da Figura 4.

- se a fórmula for $\exists \bigcirc \Psi$ retorna o conjunto de estados onde para cada estado a intersecção de seus posteriores com Sat(Ψ) for diferente de vazio, pois caso contrário, a fórmula não é verdadeira para o estado em avaliação. Pode-se visualizar este comportamento no bloco de código da linha 32 da Figura 4, onde é executado o Sat da subfórmula a esquerda na árvore sintática e, na sequência, percorre iterativamente o RSS verificando a existência de estados sucessores para o estado em avaliação. Após, realiza a intersecção com o resultado do Sat da subfórmula executado no início e, caso essa intersecção não for conjunto vazio, o estado é adicionado ao MDD de saída.
- Uma fórmula $\exists(\Phi_1 \mathbf{U} \Phi_2)$, sendo Φ_2 verdadeiro, a mesma é satisfeita. Essa definição estabelece o menor ponto-fixo que é calculado e armazenado em um conjunto T. Partindo disto, para cada estado que satisfaça Φ_1 e ainda não esteja em T, caso a intersecção de seus posteriores com T não for vazia, inclui o mesmo em T. Ao término dessa execução, retorna o conjunto T. Este comportamento é visualizado no bloco de código da linha 50 da Figura 4, onde executa o Sat de ambas as subfórmulas de EU e armazena seus resultados. Iterativamente, realiza a diferença entre os dois resultados armazenados e verifica se o resultado é diferente de vazio. Caso verdade, realiza outra iteração percorrendo todos os estados do RSS e, para cada estado, recupera seus posteriores, realiza a intersecção com o conjunto de estados resultante de Sat da subfórmula a direita, e caso não resulte vazio, acrescenta o estado neste conjunto. A execução encerra quando não houver mais estados a serem acrescentados ao MDD de saída.
- caso a fórmula for $\exists\Box\Phi$, o seu maior ponto-fixo será o conjunto de estados que satisfazem Φ, sendo estes armazenados em T. Para cada estado em T, caso a intersecção entre seus posteriores e T for vazia, remove o mesmo de T. Ao término dessa execução, retorna o conjunto T. O bloco de código da linha 91 da Figura 4 é responsável por este comportamento. A execução inicia realizando a satisfação da subfórmula de EG seguida de um laço de execução. A cada iteração, verifica se o conjunto saída, inicialmente populado pelo resultado anterior, é diferente de vazio. Caso verdade, outra iteração percorrendo o RSS é iniciada, onde para cada estado recupera seus posteriores, realiza a intersecção com o conjunto saída e, caso resulte vazio, retira o estado em avaliação deste mesmo conjunto. A execução encerra quando não mais houver estados a serem retirados.

No código da figura 4, linha 1, tem-se como parâmetro de entrada a fórmula CTL e como parâmetros de saída os ponteiros para o diagrama de decisão com o espaço de estados que satisfaz a fórmula de entrada. $ReachSS$ é também entrada para esta função mas trata-se de uma variável global já inicializada com o espaço de estados atingível do modelo (vide Geração do RSS na Figura 2). A biblioteca de diagramas de decisão oferece várias operações sobre conjuntos de estados. Assim:

- ReachSS.Levels(): retorna o número de níveis do diagrama;
- ReachSS.RootNode(Level &kp, NodeIdx &p): retorna o nível e o índice do nodo *root* do MDD que codifica o RSS;
- ReachSS.Difference(Level kp, NodeIdx p, Level kq, NodeIdx q, Level &kr, NodeIdx &r); realiza a diferença dos MDD's codificados em <kp:p> e <kq:q> e armazena no MDD codificado em <kr:r>;
- ReachSS.Set(aut_st &tuple, int value, Level &kr, NodeIdx &r): Cria um MDD em <kr:r> com o estado em "tuple";

- ReachSS.Union(Level kp, NodeIdx p, Level kq, NodeIdx q, Level &kr, NodeIdx &r): realiza a união dos MDD's codificados em <kp:p> e <kq:q> e armazena no MDD codificado em <kr:r>;
- ReachSS.Intersection(Level kp, NodeIdx p, Level kq, NodeIdx q, Level &kr, NodeIdx &r) realiza a intersecção dos MDD's codificados em <kp:p> e <kq:q> e armazena no MDD codificado em <kr:r>;
- ReachSS.NextStates(aut_st &state, Level &kr, NodeIdx &r): retorna no MDD codificado por <kr:r> todos os estados sucessores de "state";
- ReachSS.State(Level kr, NodeIdx r, bint idx, aut_st &state): dado um MDD codificado em <kr:r> e um estado em "state", retorna o índice geral do estado em "idx";

5.3. Exemplificação

Para exemplificar o uso da ferramenta, o modelo SAN referente ao problema do Jantar dos Filósofos foi utilizado, apresentando resultados de propriedades verificadas variando o número de filósofos do modelo de entrada.

Nesta exemplificação, três clássicas situações em sistemas distribuídos foram verificadas: *deadlock*, postergação indefinida e exclusão mútua. Para tanto, fez-se necessária a definição de propriedades CTL que permitam verificar tais comportamentos. A especificação e explicação dessas propriedades é encontrada na Seção 5.1. Para a realização dos testes foi utilizado um ambiente com processador Intel Core 2 Duo 2.1 GHz com 2 MB de cache L2, 3 GB de memória RAM rodando Linux Ubuntu 10.04 32 bits. Na Tabela 1 é informado o tamanho do espaço de estados atingível para cada modelo verificado enquanto que na Tabela 2 são apresentadas as propriedades verificadas bem como o resultado, tempo de verificação e memória consumida para cada caso.

Tabela 1. Espaços de Estados Atingíveis para os modelos verificados.

Quantidade de Filósofos	RSS
3	12 estados
5	70 estados
7	408 estados
9	2378 estados
11	13860 estados
13	80782 estados
14	195025 estados

As execuções da Tabela 2 demonstram que esta primeira versão da ferramenta cumpre bem seu objetivo para pequenos modelos. Já para modelos maiores se faz necessário um ambiente mais robusto.

Percebe-se o quanto oneroso o processo de verificação é analisando a execução do modelo com 14 filósofos (RSS de 195.025 estados). Verifica-se, pelas informações da Tabela 2, que algumas propriedades puderam ser verificadas no ambiente utilizado, porém a propriedade de Exclusão Mútua o verificador precisou de mais memória do que o ambiente fornecia, ocorrendo neste caso a não verificação da propriedade. O crescimento do espaço de estados aumenta significativamente o tempo e consumo de memória de uma verificação, sendo necessária a busca por opções de otimização da ferramenta e uso de máquinas com maior poder de processamento e memória.

```
void Sat(TreeNode *formula, Level &kp, NodeIdx &p){
  char op[3];
  aut_st state(ReachSS.Levels());
  Level kq, kr, ks;
  NodeIdx q, r, s;
  bool change;

  //pega o valor que esta na raiz da arvore
  strcpy(op, formula->info);
  kp = 0; p = 0;

  //verifica o valor da raiz da arvore
  switch(CtlMap.ctlMap[op]){
    case 10://caso ^, ou seja, AND
      {
        Sat(formula->left, kq, q);

        if(q != 0){
          Sat(formula->right, kr, r);
          if(r != 0)
            ReachSS.Intersection(kq, q, kr, r, kp, p);
        }
      }
        break;
      case 7://caso ¬, ou seja, negacao
          //Guarda em <kq:q> o RSS e em <kr:r>
          ReachSS.RootNode(kq, q);
          Sat(formula->left, kr, r);

          ReachSS.Difference(kq, q, kr, r, kp, p);
          break;
      case 1://caso EX, ou seja, Existe Next
          Sat(formula->left, kq, q);

          //Percorre o RSS e, pra cada estado, busca os
          //posteriores, faz a interseccao com o Sat
          //da subformula e, caso o resultado da interseccao
          //nao for vazio, adiciona o estado no DD resultado
          for(bint i=0; i<ReachSS.RSS(); i++){
            ReachSS.State(i, state);
            ReachSS.NextStates(state, kr, r);
            ReachSS.Intersection(kr, r, kq, q, kr, r);

            if(r != 0){
              ReachSS.Set(state, 1, kr, r);
              ReachSS.Union(kp, p, kr, r, kp, p);
            }
          }
          break;
      case 8://caso EU, ou seja, Existe Until
        {
          Sat(formula->left, kq, q);
          Sat(formula->right, kp, p);

          //Enquanto houver estados para adicionar
          //continua no laco
          change = true;
          while(change){
            change = false;

            //Realiza a diferenca entre Sat de phi1 e o conjunto T
            //(codificado por <kp:p> onde T eh o conjunto que
            //inicialmente representa phi2 e incrementado com
            //novos estados cada vez que o estado avaliado
            //satisfaz uma condicao, sendo T o retorno)
            ReachSS.Difference(kq, q, kp, p, kr, r);

            //Caso a diferenca seja diferente de vazio,
            //percorre os estados de RSSe, para cada
            //estado, busca os posteriores e realiza
            //a interseccao deles com T
            if(r != 0){
              for(bint idx=0; idx<ReachSS.NumStates(kr, r); idx++){
                ReachSS.MTMDD::State(kr, r, idx, state);
                //Post(s) guarda em <ks:s>
                ReachSS.NextStates(state, ks, s);
                ReachSS.Intersection(ks, s, kp, p, ks, s);

                //Caso a interseccao dos posteriores com T seja
                //diferente de vazio inclui o estado em avaliacao em T
                if(s != 0){
                  ReachSS.Set(state, 1, ks, s);
                  ReachSS.Union(kp, p, ks, s, kp, p);
                  change = true;
                }
              }
            }
          }
        }
        break;
      case 2://caso EG, ou seja, Existe Globally
        Sat(formula->left, kp, p);

        //Enquanto houver estados para adicionar continua no laco
        change = true;
        while(change){
          change = false;

          //Caso T seja diferente de vazio, percorre os estados de RSS
          //e, para cada estado, busca os posteriores
          //e realiza a interseccao deles com T
          if(p != 0){
            for(bint idx=0; idx<ReachSS.NumStates(kp, p); idx++){
              ReachSS.MTMDD::State(kp, p, idx, state);
              //Post(s) guarda em <kq:q>
              ReachSS.NextStates(state, kq, q);
              ReachSS.Intersection(kq, q, kp, p, kq, q);

              //Caso a interseccao dos posteriores com T seja vazio
              //retira de T o estado em avaliacao
              if(q == 0){
                ReachSS.Set(state, 1, kq, q);
                ReachSS.Difference(kp, p, kq, q, kp, p);
                change = true;
              }
            }
          }
        }
        break;
      default:
        if(strcmp(op, "T") == 0){ //caso T, ou seja, true
          //Pega o root do DD que codifica o RSS
          ReachSS.RootNode(kp, p);
        }else{
          //Caso nao seja nenhum operacao,
          //entao e  uma proposição atômica
          int value = atoi(op);
          kp = kFunc[value];
          p = pFunc[value];
        }
        break;
  }
}
```

Figura 4. Código-fonte da rotina SAT.

6. Conclusão

Apresentamos neste artigo o estado da arte atual do desenvolvimento de um verificador simbólico de modelos descritos em Redes de Autômatos Estocásticos.

Tabela 2. Exemplificação de Verificação de Modelos e Propriedades.

Modelo: Jantar dos Filósofos	Tempo (s)	Memória
Postergação Indefinida (Satisfeita)		
3 Filósofos	0.035	836 KB
5 Filósofos	0.045	1.4 MB
7 Filósofos	0.075	4.8 MB
9 Filósofos	0.70	27.6 MB
11 Filósofos	4.43	170.5 MB
13 Filósofos	28.63	809.7 MB
14 Filósofos	300.38	2.2 GB
Exclusão Mútua (Insatisfeita. Modelo mutuamente exclusivo)		
3 Filósofos	0.035	832 KB
5 Filósofos	0.043	1.45 MB
7 Filósofos	0.085	5.0 MB
9 Filósofos	0.64	29.8 MB
11 Filósofos	3.38	192.9 MB
13 Filósofos	59.29	1.2 GB
14 Filósofos	-	-
Deadlock (Insatisfeita. Não há deadlock)		
3 Filósofos	0.032	844 KB
5 Filósofos	0.042	1.4 MB
7 Filósofos	0.090	4.3 MB
9 Filósofos	0.40	18.9 MB
11 Filósofos	2.83	96.9 MB
13 Filósofos	21.52	551.1 MB
14 Filósofos	43.37	675.4 MB

Em uma realidade em que sistemas computacionais estão presentes nas mais diversas aplicações, técnicas de validação de sistemas, como a verificação de modelos, são importantes para garantir propriedades fundamentais para estes sistemas, como disponibilidade, segurança de funcionamento (*safety*), entre outros. Neste contexto, a ferramenta em desenvolvimento contribui para verificação dessas propriedades em modelos de sistemas descritos em SAN, formalismo ainda não contemplado por nenhum verificador.

Através dos testes apresentados na Seção 5.3, pode-se concluir que o desempenho da ferramenta ficou limitado a modelos pequenos, principalmente no que tange a memória necessária, visto que o aumento no número de estados acarreta em um crescimento exponencial do consumo de memória pelo verificador.

7. Trabalhos Futuros

Como trabalhos futuros, objetiva-se o desenvolvimento e geração de contra-exemplos para a ferramenta em construção. Isto se faz necessário dada a importância que os mesmos apresentam para verificação de modelos e pelo poder que tem em descrever um comportamento incorreto, permitindo identificar a falha na modelagem. Para otimização da ferramenta, objetiva-se o estudo e implementação da paralelização do algoritmo de satisfação de fórmulas, o emprego da técnica de Saturação para *Model Checking* e a utilização de mecanismos de *Garbage Collection*.

Referências

Baier, C. and Katoen, J.-P. (2008). *Principles of Model Checking*. MIT Press.

Benoit, A., Brenner, L., Fernandes, P., Plateau, B., Sbeity, I., and Stewart, W. J. (2007). Peps 2007 user manual.

Benoit, A., Brenner, L., Fernandes, P., Plateau, B., and Stewart, W. J. (2003). The peps software tool. In *Computer Performance Evaluation/TOOLS*, pages 98–115.

Brenner, L., Fernandes, P., Plateau, B., and Sbeity, I. (2007). Peps 2007 - stochastic automata networks software tool. *Quantitative Evaluation of Systems, International Conference on*, 0:163–164.

Brenner, L., Fernandes, P., and Sales, A. (2005). The Need for and the Advantages of Generalized Tensor Algebra for Kronecker Structured Representations. *International Journal of Simulation: Systems, Science & Technology (IJSIM)*, 6(3-4):52–60.

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

Czekster, R. (2010). *Solução numérica de descritores markovianos a partir de reestruturações de termos tensoriais*. PhD thesis, Pontifícia Universidade Católica do Rio Grande do Sul, Brasil.

Fernandes, P., Plateau, B., and Stewart, W. J. (1998). Efficient descriptor-vector multiplication in Stochastic Automata Networks. *Journal of the ACM*, 45(3):381–414.

Hadzic, T., Hooker, J. N., O'Sullivan, B., and Tiedemann, P. (2008). Approximate compilation of constraints into multivalued decision diagrams. In *Proceedings of the 14th international conference on Principles and Practice of Constraint Programming*, CP '08, pages 448–462, Berlin, Heidelberg. Springer-Verlag.

Hurth, M. and Ryan, M. (2004). *Logic In Computer Science*. Cambridge University Press.

Miller, D. and Drechsler, R. (1998). Implementing a multiple-valued decision diagram package. In *28th IEEE International Symposium on Multiple-Valued Logic*, pages 52–57.

Plateau, B. (1985). On the stochastic structure of parallelism and synchronization models for distributed algorithms. *SIGMETRICS Perform. Eval. Rev.*, 13:147–154.

Sales, A. and Plateau, B. (2009). Reachable state space generation for structured models which use functional transitions. In *6th International Conference on the Quantitative Evaluation of Systems (QEST'09)*, pages 269–278, Budapest, Hungary. IEEE Computer Society.

Stewart, W. J. (2009). *Probability, Markov Chains, Queues, and Simulation: The Mathematical Basis of Performance Modeling*. Princeton University Press, USA.

XIII Workshop de Testes e Tolerância a Falhas

♦

Sessão Técnica 4

Cobertura de Falhas e Replicação na Cloud

Detecção de Falhas em Autômatos Grid

Gilson Doi Junior[1], **Adilson Luiz Bonifácio**[1]

[1]Departamento de Computação – Universidade Estadual de Londrina (UEL)
Caixa Postal 6.001 – 86.051-980 – Londrina – PR – Brazil

doijunior@gmail.com, bonifacio@uel.br

Abstract. *Real-time systems are, in general, critical systems interacting with the environment through I/O events in which time restrictions are deemed important. Model-based testing is a widely used rigorous approach to provide more accuracy on testing real-time systems. Formal techniques that can handle continuous time evolution is a challenge. Therefore several approaches rely on discretization methods to obtain discretized models, called grid automata, corresponding to original timed models. Test suites can be extracted from grid automata to verify whether given implementations adhere to the specification. However a very important issue on testing real-time systems is the fault coverage. In this work we address a fault coverage model for grid automata following the fault models for finite state machines and timed models.*

Resumo. *Sistemas de tempo real são, em geral, sistemas críticos que interagem com o ambiente externo através de eventos de E/S onde as restrições de tempo são consideradas importantes. Teste baseado em modelos é uma abordagem rigorosa amplamente usada para dar mais precisão no teste de sistemas de tempo real. Técnicas formais que podem lidar com a evolução contínua do tempo é um desafio. Por isso, várias abordagens contam com métodos de discretização para obter modelos discretizados, chamados autômatos grid, correspondentes aos modelos temporizados originais. Conjuntos de teste podem ser extraídos a partir de autômatos grid para verificar se implementações dadas se aderem à especificação. Contudo, uma questão muito importante no teste de sistemas de tempo real é a cobertura de falhas. Neste trabalho abordamos um modelo de cobertura de falhas para autômatos grid seguindo os modelos de falha para máquinas de estados finitos e modelos temporizados.*

1. Introdução

Os avanços tecnológicos de hardware e software tem possibilitado uma maior adesão às técnicas de automatização de tarefas relacionadas ao desenvolvimento de sistemas. Entre esses sistemas estão os chamados sistemas críticos onde a ocorrência de uma falha pode resultar em prejuízos financeiros e danos irreparáveis. Técnicas rigorosas, em especial na fase de teste, são de extrema importância para se minimizar a ocorrência de falhas em sistemas dessa natureza.

Teste baseado em modelos [M. Blackburn 2004, Utting and Legeard 2007] é uma abordagem rigorosa de teste que se apoia em métodos e modelos formais. Sua principal característica é a utilização de uma estrutura formal para especificar sistemas e extrair conjuntos de teste capazes de identificar o maior número de falhas possíveis. Porém,

ainda existem muitos desafios na utilização das técnicas de teste baseado em modelos, apesar de se tratar de um assunto bem estabelecido e com vários estudos já realizados. Um desses desafios é lidar com sistemas de tempo real [Alur 1999, Alur and Dill 1994], onde o instante de tempo influencia o comportamento e determina a ocorrência das ações do sistema. Lidar com o aspecto de tempo contínuo usando modelos formais se torna ainda mais difícil quando o foco é a atividade de teste para tais sistemas.

Alguns trabalhos nessa linha [En-Nouaary et al. 2002, Springintveld et al. 2001, En-Nouaary and Hamou-Lhadj 2008, Bonifácio and Moura 2011] focam técnicas de discretização para lidar com a evolução contínua de tempo. Os modelos discretizados, também chamados de grid, capturam aspectos de tempo, permitindo a análise e detecção de falhas importantes dos sistemas.

Já as estratégias de detecção de falhas podem ser avaliadas conforme os diferentes objetivos de teste, indicando qual a cobertura de falhas provida. As avaliações podem ser qualitativas, através de propostas de teste [Bonifácio and Moura 2011, Weiglhofer and Wotawa 2009], ou quantitativas, através de modelos de falha [Chow 1978, En-Nouaary et al. 1999].

Este trabalho pretende analisar a cobertura de falhas para autômatos grid obtidos conforme técnicas mais recentes de discretização [Bonifácio and Moura 2011, Bonifacio and Moura 2009]. Em geral, autômatos grid recaem no problema de explosão combinatória de estados, comumente encontrado em abordagens que tratam de sistemas temporizados. Grids mais gerenciáveis, então, permitem o estudo e a análise da cobertura de falhas para sistemas de tempo real. A estrategia para se identificar falhas no modelo grid se baseia nos modelos de falha propostos para as Máquinas de Estados Finitos (MEFs) [Chow 1978] e para os *Timed Input/Output Automata* (TIOA) [En-Nouaary et al. 1999].

Este trabalho está organizado da seguinte forma. A Seção 2 apresenta a abordagem de teste baseado em modelos e alguns formalismos relacionados a proposta. Trabalhos relacionados neste contexto sobre teste em sistemas de tempo real são descritos na Seção 3. A Seção 4 apresenta a noção de conformidade e as importantes relações conhecidas para modelos convencionais e temporizados. O conceito de modelo de falhas, bem como os modelos de falha específicos usados como base neste trabalho são apresentados na Seção 5. A Seção 6 descreve uma análise preliminar da capacidade de detecção de falhas em modelos discretizados. As considerações finais e direções futuras de trabalho são apresentadas na Seção 7.

2. Teste Baseado em Modelos

Abordagens tradicionais de teste, em geral, modelam sistemas através de métodos convencionais, incluindo nesses casos o apoio efetivo de linguagem natural. Esse tipo de abordagem possibilita interpretações ambíguas, podendo resultar em produtos com erros. Para se evitar tais ambiguidades as técnicas de teste baseado em modelos utilizam linguagens bem definidas, como o UML [Utting and Legeard 2007], ou modelos formais com forte embasamento matemático. Formalismos são adotados para retirar a ambiguidades do modelo de especificação e possibilitar os métodos de geração de casos de teste. Essa abordagem diminui drasticamente a possibilidade de erros de interpretação, principalmente no levantamento de requisitos [Hierons et al. 2009].

Outro problema encontrado nas metodologias tradicionais é a seleção do conjunto de testes. Muitas vezes essa seleção é realizada de maneira manual e arbitrária, não havendo uma análise mais precisa da eficiência e da eficácia de detecção das falhas. O modelo formal de especificação facilita a análise da estrutura do sistema, mostrando a cobertura provida pela conjunto de testes, além de possibilitar técnicas que automatizem a extração e execução desses testes. Utilizando modelos formais para especificar requisitos de sistema e definir uma rotina rigorosa de extração e execução do conjunto de testes é possível reduzir consideravelmente o número de falhas não detectadas, proporcionando uma maior confiabilidade aos métodos [M. Blackburn 2004].

Existem diversos métodos e técnicas que seguem essa abordagem de teste, porém divergindo em diferentes aspectos, como no modelo formal adotado ou nas características da rotina de extração e execução dos testes. Apesar dessas diferenças, as atividades de teste baseado em modelos podem ser definidas em algumas etapas: (i) modelagem dos requisitos da especificação do sistema; (ii) extração dos conjuntos de teste a partir do modelo de especificação; e (iii) o veredito de conformidade, obtido pela comparação do comportamento da implementação em relação à especificação.

Um dos desafios encontrados na utilização dessa abordagem é a sua adoção em sistemas de tempo real, devido a dificuldade tanto para se modelar sistemas com tempo contínuo quanto para lidar com aspectos temporais em aplicações práticas. A seguir são apresentados os principais modelos formais relacionados com este trabalho.

2.1. Formalismos

Os modelos formais são mecanismos adotados para especificar sistemas de maneira bem definida, utilizados para auxiliar as atividades de teste baseado em modelos.

Existe uma grande variedade de modelos, cada uma com suas próprias características que influenciam na sua capacidade de abstração e expressão dos sistemas, bem como nas restrições impostas. Alguns tipos específicos de sistemas, tais como os sistemas de tempo real e/ou reativos, necessitam de um poder de representatividade maior para modelar suas particularidades, como a evolução contínua do tempo e as respostas a estímulos externos do sistema, respectivamente.

Máquinas de Estados Finitos O modelo MEF é utilizado para especificar sistemas que reagem a estímulos do ambiente externo produzindo respostas [Chow 1978]. Neste caso, os estímulos externos e os eventos autônomos gerados pelo sistema, são entradas e saídas associadas e representadas numa transição da máquina. A seguir, uma MEF é definida, formalmente, como uma tupla $M = (X, Y, S, s_0, \delta, \lambda)$, onde:

- X é o conjunto finito e não vazio de símbolos do alfabeto de entrada;
- Y é o conjunto finito e não vazio de símbolos do alfabeto de saída;
- S é o conjunto finito e não vazio de estados;
- s_0 é o estado inicial da máquina;
- δ é a função de mudança de estado, onde $\delta : (X \times S_1) \rightarrow S_2$;
- λ é a função de saída, onde $\lambda : (X \times S) \rightarrow Y$.

Timed Input/Output Automata O modelo TIOA é usado para especificar sistemas temporizados, com a ocorrência de saídas autônomas produzidas pelo sistema [Kaynar et al. 2006]. Basicamente, um TIOA é definido através de um BTDA[1] o

[1] *Bounded Time Domain Automata*

qual inclui uma descrição semântica da evolução contínua de tempo para modelar sistemas de tempo real [Alur and Dill 1994]. Um BTDA é, formalmente, definido pela tupla $M = (S, s_0, \Sigma, C, \nu_0, Inv, T)$, contendo

- um conjunto finito de estados S;
- um estado inicial s_0;
- um conjunto finito de ações Σ;
- um conjunto de relógios C, onde uma variável relógio $c \in C$ descreve a evolução contínua do tempo;
- uma interpretação inicial de relógios ν_0, $\nu(c) = 0$ para todo $\nu(c) \in \nu_0$. Uma interpretação de relógio ν é o conjunto de interpretações $\nu(c)$,$\forall c \in C$ que descreve o valor de cada relógio em um instante de tempo;
- uma função $Inv : S \rightarrow \Phi_C$ que mapeia cada condição de relógio de um estado. Uma condição de relógio é uma expressão booleana obtida através de combinação das formas atômicas $c < m$, $c = m$ e $c > m$, onde c é um relógio e $m \in \mathbb{Q}_{\geq}$. Essas expressões determinam quando uma transição pode ser executada.
- um conjunto de transições de estado $T \subseteq (S \times \Sigma \times \Phi_C \times [C \curvearrowright \mathbb{Q}_{\geq}] \times S)$.

O modelo TIOA nada mais é que uma variação do modelo BTDA, com um particionamento no conjunto de ações em subconjuntos de entrada e de saída. Formalmente, um TIOA é representado pela tupla $M = (B, X, Y)$, onde:

- $B = (S, s_0, \Sigma, C, \nu_0, Inv, T)$ é um BTDA;
- $X \subseteq \Sigma$ e $Y \subseteq \Sigma$ são os conjuntos finitos de ações de entrada e saída, respectivamente, onde $X \cap Y = \emptyset$;

Autômato grid O autômato grid é uma representação discretizada do modelo TIOA. Essa discretização é obtida através da utilização de uma granularidade escolhida. Logo, a obtenção de um grid depende fortemente da técnica de discretização proposta para o modelo temporizado [Bonifácio and Moura 2011]. Um grid pode ser definido pela tupla $M = (S, s, \Sigma, T)$, onde:

- S é um conjunto finito de estados;
- s é o estado inicial;
- $\Sigma \cup g$ é um conjunto finito de rótulos, onde Σ é o alfabeto de entrada e saída do modelo de tempo, e g a granularidade da discretização;
- $T \subseteq (S \times (\Sigma \cup g) \times S)$ é um conjunto finito de transições.

Este trabalho se baseia em métodos recentes de discretização, onde a obtenção de grids com espaço de estado mais compactos diminui o problema da explosão combinatória [Bonifácio and Moura 2011]. Isso permite que a análise de cobertura de falhas proposta para grids seja aplicada em situações práticas.

3. Trabalhos Relacionados ao Teste em Sistemas de Tempo Real

Sistemas de tempo real são caracterizados pela execução de ações onde os instantes de tempo em que ocorrem são importantes. Alguns formalismos específicos são utilizados para modelar a passagem de tempo. Porém, a característica contínua do tempo impossibilita na maioria das vezes sua manipulação em aplicações práticas. Uma das estrategias é a discretização do tempo contínuo dos modelos formais utilizando granularidades adequadas de tal forma que ainda possibilitem a exploração dos requisitos de tempo definidos pelo sistema.

Alur e Dill [Alur 1999, Alur and Dill 1994] propõem uma abordagem de discretização utilizando uma granularidade baseada no número de regiões de relógio do modelo do sistema. Uma região de relógio define um conjunto de valorações dada através das restrições impostas aos relógios do modelo [Alur 1999, En-Nouaary et al. 2002]. Esse número de regiões cresce de acordo com o número de relógios e valores limites que cada relógio atinge. Uma granularidade muito fina pode, então, resultar num crescimento exponencial no número de estados do sistema tornando inviável sua utilização.

En-Nouaary [En-Nouaary et al. 1999] propôs um modelo de falhas para TIOA utilizando autômato de região. Springintveld [Springintveld et al. 2001] e En-Nouaary [En-Nouaary et al. 2002] também utilizam uma granularidade baseada em regiões de relógio para propor variações dos métodos W [Chow 1978] e Wp [Fujiwara et al. 1991] adaptados para sistemas de tempo real modelados em TIOA. A versão original desses métodos proporcionam cobertura total de falhas para modelos de MEF. Em suas aplicações tradicionais para o modelo de MEF, tanto o método W quanto o método Wp proporcionam uma cobertura total das falhas. Porém, em sistemas de tempo real, conforme o número de relógios aumenta, esses métodos se tornam inviáveis na prática.

Em outro trabalho En-Nouaary [En-Nouaary and Hamou-Lhadj 2008] utiliza uma abordagem distinta que explora a estrutura do TIOA. Cada transição é verificada apenas nos limites superior e inferior do intervalo de cada condição de relógio, bem como no ponto médio desses limites. Isso possibilita lidar com o modelo original sem a necessidade de se obter uma representação discretizada do mesmo. Contudo, os três pontos observados em cada transição não oferecem uma representação razoável do tempo contínuo sobre todo intervalo de habilitação da transição. Logo, não existe nenhuma garantia sobre a ocorrência ou não de falhas que possam ocorrer nas lacunas de tempo.

Em outra abordagem, Bonifacio e Moura [Bonifácio and Moura 2011] propõem um *framework* de teste que permite uma discretização mais flexível de um TIOA. A escolha da granularidade pode então ser mais grossa, permitindo a obtenção de um espaço de estado mais gerenciável. Porém, a eficácia do método é demonstrada através de propostas de teste, o que restringe a conjunto de falhas detectáveis.

4. Relações de Conformidade

Uma relação de conformidade verifica se uma implementação se comporta de acordo com a sua especificação. A abordagem mais comum é determinar uma relação de conformidade através de propriedades que devem ser satisfeitas quando uma implementação está de acordo com a especificação.

Algumas técnicas de teste definem que uma implementação está conforme sua especificação se elas sempre se comportam da mesma maneira. Quando essa condição não é totalmente verificável, técnicas utilizam hipóteses que limitam o escopo do comportamento a ser verificado. Alguns exemplos de hipóteses que restringem o escopo de teste são: utilizar o mesmo alfabeto de entrada/saída da especificação; limitar o número de estados a ser considerado nas implementações candidatas; e operações de reinicialização do sistema em todo estado da implementação [En-Nouaary et al. 2002].

A seguir são descritas algumas importantes relações de conformidade.

4.1. *Input-Output Conformance*

A relação *input-output conformance* (ioco) é definida originalmente para implementações dadas por *Input Output Transition Systems* (IOTSs) [Tretmans 2008] e especificações modeladas em *Labelled Transition Systems* (LTSs) [Tretmans 2008].

Essa relação se baseia nas sequências de entrada aplicadas ao sistema, também chamadas de traços. O comportamento observável em resposta a um traço é chamado de saída. A relação ioco define que uma implementação I está em conformidade com uma especificação E, se ao aplicar cada traço de E em I a saída obtida é a mesma observada em E [Tretmans 2008].

Considerando que o conjunto de todo par traço/saída descreve o comportamento do sistema, uma das relações ioco pode ser definida como $I\ ioco\ E \equiv comportamento(E) \subseteq comportamento(I)$.

4.2. *Timed Input-Output Conformance*

A relação *timed input-output conformance* (tioco) é uma extensão da relação ioco com o objetivo de capturar sistemas de tempo real modelados com TIOA. Como sistemas de tempo real são ditados pelo tempo contínuo é necessário que os instantes em que as ações ocorrem sejam identificados. *Delays* são utilizados para descrever os intervalos de tempo decorrido. Logo, os traços e saídas possuem *delays* em suas sequências para identificar intervalos entre as ocorrências de ações.

Dados uma implementação I e uma especificação E, a relação de conformidade tioco é definida de maneira similar ao ioco: $I\ tioco\ E \equiv comportamento(E) \subseteq comportamento(I)$. Porém, neste caso os traços são compostos além das entradas e das saídas, também pelos instantes de tempo em que essas ações ocorrem. Algumas propostas de extração de casos de teste para TIOA utilizam *ticks* de tempo [Krichen and Tripakis 2004]. Na prática essas técnicas acabam se comportando como os modelos discretizados descritos anteriormente, podendo ser considerado um caso particular.

5. Detecção de falhas

Avaliar a capacidade de detecção de falhas de um conjunto de testes observando apenas o número de falhas encontradas é uma métrica grosseira, pois diferentes conjuntos de testes podem exercitar diferentes aspectos da especificação. Métricas mais eficientes focam a amplitude da estrutura do sistema explorado durante a execução dos testes e a capacidade de se exercitar partes específicas em busca de falhas.

Duas abordagens se destacam para essa finalidade, uma com o foco na detecção de falhas específicas, chamada proposta de teste [Bonifácio and Moura 2011, Weiglhofer and Wotawa 2009], e outra que foca na estrutura da especificação e classes de falha, chamada modelo de falhas [Chow 1978, En-Nouaary et al. 1999].

A proposta de teste é considerada uma abordagem qualitativa, pois verifica se o conjunto de testes é capaz de detectar propriedades de falha (ou desejáveis) da implementação, não levando em conta uma relação mais precisa sobre o número de falhas detectadas. O formalismo adotado para especificar o sistema também é utilizado para modelar as propriedades desejáveis ou de falha a serem verificadas. Esse modelo então

representa um comportamento que através de sequências de entradas e saídas observáveis levam o controle do modelo aos estados de falha (ou desejáveis) do sistema.

No modelo de falha se observa a estrutura do sistema e as potenciais falhas que implementações candidatas podem apresentar. Através do modelo de falhas é possível identificar classes que o conjunto de testes é capaz de identificar. Dessa forma, uma análise quantitativa sobre a capacidade de detecção pode ser realizada.

Sabe-se que na abordagem de teste baseado em modelos não se conhece qualquer informação a respeito da estrutura da implementação. No entanto, assume-se que a implementação candidata é modelada com o mesmo formalismo utilizado para representar a especificação. Por isso, o modelo de falhas é caracterizado através das possíveis mutações que a especificação pode apresentar em relação ao modelo original.

Nas próximas subseções são descritos os modelos de falha propostos para os modelos MEF e TIOA.

5.1. Modelo de falhas para MEF

Chow [Chow 1978] apresenta um método de geração de casos de teste para modelos MEF e propõe um modelo de falhas que divide as potenciais falhas em três grupos:

Falhas de operação: ocorre quando uma função de operação apresenta uma saída distinta da especificada.

Falhas de transferência: ocorre quando uma função de transferência leva a execução de uma transição a um estado distinto do especificado.

Falhas de estados extras/ausente: ocorre quando a implementação não está em conformidade com a implementação, mas é possível corrigir isso incluindo ou retirando estados.

As falhas de operação são detectadas pelo conjunto de testes ao exercitar todas as transições do modelo de especificação. As falhas de transferência são detectadas pela utilização de um conjunto de sequências de entrada, também chamado de conjunto de identificação, capaz de identificar cada estado do modelo de especificação.

A proposta não prevê falhas de transições extras, pois por hipótese assume-se MEFs completamente especificadas. Página Cada estado de uma MEF completamente especificada possui transições para cada símbolo do alfabeto de entrada. Outra hipótese assumida é que todo estado do modelo é alcançável a partir do estado inicial. Assim falhas de estados ausentes podem ser detectadas exercitando transições que levem ao estado ausente na implementação, verificando se uma falha de transferência não ocorre.

Como estados extras não tem seu comportamento especificado, esses devem ser testado para cada símbolo possível do alfabeto de entrada. Quando estados extras são permitidos, a verificação é estendida por sequências de maior comprimento proporcional ao número de estados extras suposto.

5.2. Modelo de falhas para TIOA

En-Nouaary [En-Nouaary et al. 1999] propõe um modelo de falha para TIOA, classificando as potenciais falhas em: falhas de tempo, falhas de ação e falhas de transferência de estados.

As falhas de tempo são aquelas decorrentes de alterações relacionadas a semântica de tempo, como as restrições e reinicializações de relógio. Contudo, existem modificações que mantêm o comportamento da implementação equivalente ao da especificação, pois representam falhas não efetivas. A seguir são descritas as falhas relacionadas ao tempo.

Falha de reinicialização de relógio: ocorre quando o conjunto de relógios a ser reinicializado não é implementado corretamente. Esse tipo de falha se divide em dois subcasos:

- na ausência da reinicialização dos relógios o tempo continua a evoluir alcançando valores não previstos na especificação.
- na implementação de uma reinicialização inexistente o relógio passa a não explorar valores limites do sistema.

Falha de restrição da condição de relógio: ocorre quando uma transição com ação de entrada é implementada com uma condição de relógio mais restrita, limitando o intervalo de execução da ação. No caso de ações de saída, que não são ações controláveis, a restrição da condição é considerada válida uma vez que o intervalo restrito está contido no intervalo especificado.

Falha de relaxamento da condição de relógio: Ao relaxar uma condição de relógio, a ocorrência de uma ação é aceita num intervalo maior que o especificado. No caso de uma ação de saída, a ocorrência além do intervalo da condição de relógio não é permitida pela especificação, mesmo as saídas não sendo controláveis. Por isso, esse tipo de falha ocorre tanto em transições com ações de entrada como de saída, diferente das falhas de restrição da condição de relógio.

Falhas de ação e transferência de estados não dependem da semântica de tempo, se comportando similarmente ao modelo de falhas para MEF. A falha de ação ocorre quando uma transição é implementada com uma ação diferente da especificada. Essa falha é similar a falha de operação, contudo no modelo MEF cada transição disparada possui uma entrada e uma saída associada. No TIOA a transição possui uma única ação, seja de entrada ou de saída. Já a falha de transferência ocorre quando uma transição é implementada com o estado destino diferente do especificado.

O modelo de falhas para TIOA foi proposto para analisar a detecção de falhas para conjuntos de teste obtidos com o método *Timed Wp* [En-Nouaary et al. 2002]. Essa análise se baseia na representação de autômatos região. A ocorrência de falhas efetivas descritas no modelo modifica a estrutura do autômato de região [Alur and Dill 1994] que modela a implementação. Essas alterações podem ser no número de estados e transições, ou no estado destino de transições. Como a estrutura de um autômato de região é composta por estados e transições rotuladas, similarmente a estrutura do modelo MEF, essas alterações podem ser mapeadas através do modelo de falhas para MEF.

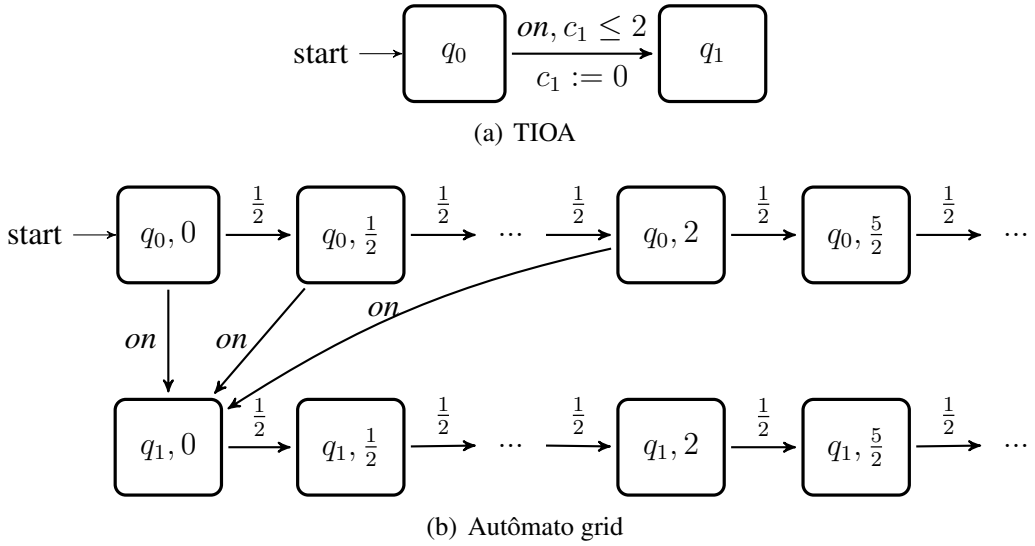
6. Identificação de Falhas em Autômatos Grid

Nesta seção apresentamos a análise realizada sobre a estrutura de autômatos grids e da identificação de suas classes de falha. O objetivo é verificar a capacidade de detecção de falhas de um conjunto de testes para TIOA com base em seus autômatos grid correspondentes.

Com o intuito de ilustrar a identificação dessas classes de falha num autômato grid, assumimos o TIOA especificação e sua correspondente discretização representados

na Figura 1. O TIOA possui uma transição com a ação *on* a partir do estado inicial, que pode ser executada quando o relógio c_1 possui interpretação menor ou igual a 2. Além disso, através dessa transição o relógio c_1 é reinicializado com valor 0.

Nas subseções a seguir, o objetivo é analisar cada classe de falha em TIOA através do autômato grid e propor alternativas sistemáticas para detectá-las.



(a) TIOA

(b) Autômato grid

Figura 1. TIOA e seu grid correspondente

6.1. Falha de ação

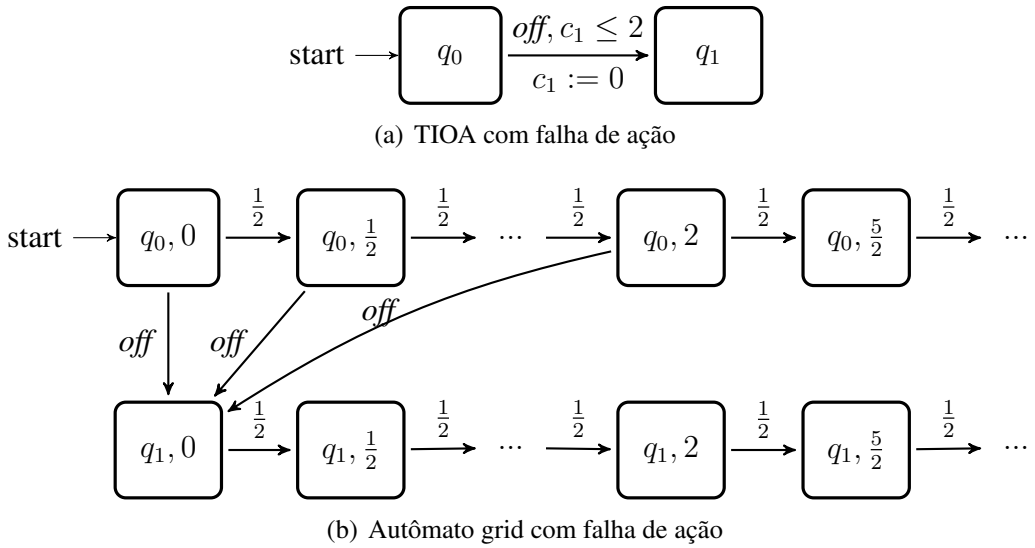
A execução de uma transição num TIOA pode ocorrer em qualquer instante de tempo no intervalo habilitado pela condição de relógio. No autômato grid cada transição do TIOA é simulada por uma transição em cada estado discretizado associado ao estado original do TIOA que habilita sua execução, como mostra a Figura 1.

Uma implementação apresenta falha de ação num TIOA quando uma transição apresenta uma ação distinta da especificação. A implementação candidata da Figura 2(a) para o TIOA da Figura 1(a) apresenta uma falha de ação executando a transição do sistema para uma ação *off* que é diferente da ação *on* no modelo da especificação. Por isso, todas as transições do autômato grid que simulam a transição com falha do TIOA no autômato grid devem ser rotuladas com a ação de falha, como mostra a Figura 1(b) para a ação *off*.

Como essa classe de falhas não depende do tempo, seu comportamento no autômato grid é similar a falha de operação no modelo MEF. Se a especificação for determinística, o estado fonte de uma transição implementada com falha de ação, deixa de responder a ação especificada. Assim um conjunto de testes que exercita cada transição do autômato grid da especificação é capaz de apontar essas falhas pela divergência entre as ações da implementação e da especificação.

6.2. Falha de transferência

Falhas de transferência são caracterizadas por uma transição com estado alvo distinto do especificado. Quando o estado alvo da transição do TIOA muda, o estado alvo de cada transição que a simula no autômato grid também é alterado. Logo, todas as transições



(a) TIOA com falha de ação

(b) Autômato grid com falha de ação

Figura 2. TIOA com falha de ação e seu grid correspondente

do autômato grid que simulam uma transição do TIOA que implementada uma falha de transferência apresentarão a mesma falha.

Essa classe de falha, assim como a falha de ação, não depende da evolução do tempo. A forma de detecção segue de maneira similar àquela utilizada para o modelo MEF. Alguns métodos conhecidos, como o W e o Wp, utilizam conjuntos de identificação [Chow 1978, Fujiwara et al. 1991] que, concatenados a cada caso de teste, são capazes de identificar se o estado alvo da sequência de transições é implementado de maneira correta.

6.3. Falha de condição de relógio

O relaxamento ou restrição das condições de relógio resultam em falhas na implementação à medida que o intervalo de habilitação da transição é alterado. Na simulação em autômato grid essa mudança é refletida na ausência ou ocorrência de transições extras.

Cada estado do TIOA é simulado no autômato grid por um conjunto de estados rotulados pelo par (q, ν), onde ν é uma interpretação de relógio discretizada para o estado original q do TIOA. Quando esse estado é a origem de uma transição do TIOA, cada estado do grid no subconjunto de estados que simula o intervalo onde a condição de relógio está habilitada tem uma transição rotulada simulando a transição do TIOA para o determinado instante de tempo acumulado no intervalo.

Quando ocorre uma falha de restrição da condição de relógio, estados discretizados do autômato grid deixam de habilitar a condição de relógio. Por exemplo, caso uma condição mais restrita, $c_1 < 2$, seja implementada no exemplo da Figura 1(a), o estado do grid com interpretação de relógio $c_1 = 2$ não habilita a condição necessária para executar a transição. Como consequência a transição a partir desse estado que simula esse instante de tempo é ausente, como mostra a Figura 3(a).

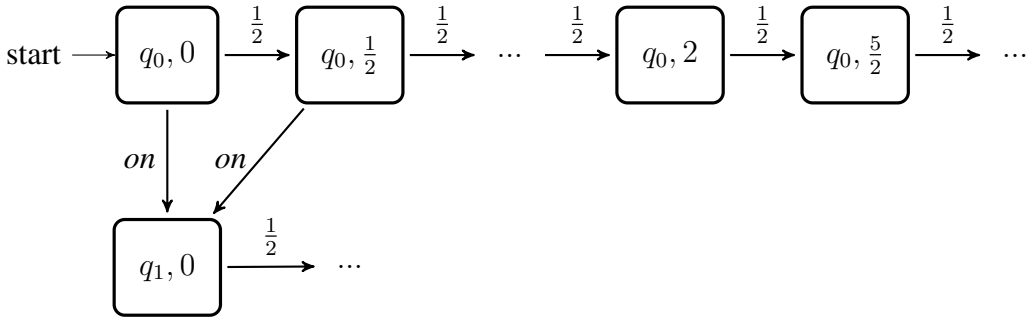
Quando ocorre uma falha de relaxamento da condição de relógio, mais estados passam a habilitar tal condição. Para ilustrar este caso, assuma uma condição mais re-

laxada, $c_1 < 3$", implementada no exemplo da Figura 1(a). O estado com interpretação de relógio $c_1 = \frac{5}{2}$ habilita a condição necessária para executar a transição. Como consequência o estado que simula esse instante de tempo apresenta uma transição extra, como mostra a Figura 3(b).

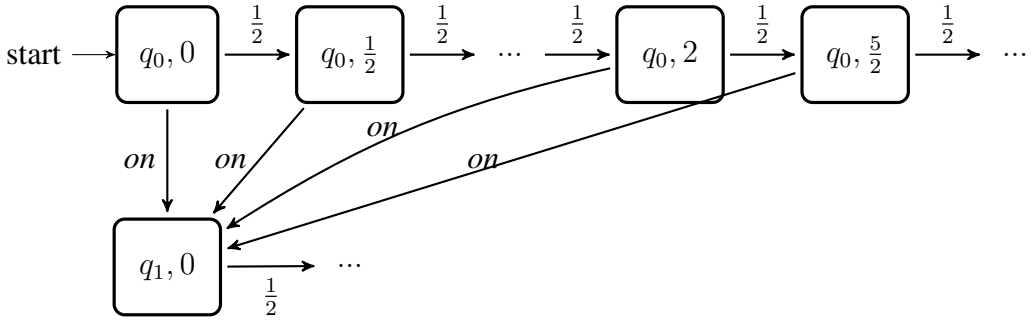
Um conjunto de testes que exercita todas as transições do grid da especificação, detecta a falha de restrição da condição de relógio já que a transição ausente na implementação não pode ser exercitada.

Uma abordagem para detecção de falhas de relaxamento da condição de relógio pode ser obtida através da hipótese em que implementações podem ter transições extras. Essa estrategia é similar à detecção de estados extras para o modelo MEF, pois, da mesma forma, não existe informação sobre o comportamento e a existência dessas transições extras provenientes da especificação.

Outra abordagem para lidar com transições extras, também utilizada no modelo MEF, é assumir que a implementação é determinística e que a especificação é completa. No caso de transições extras, geradas por relaxamento da condição de relógio numa transição, vale lembrar que ações de saída não são controláveis. Por isso, não há garantias de que toda transição de saída será exercitada durante a execução dos testes. Uma implementação com falha de relaxamento da condição de relógio numa transição de saída pode então apresentar um comportamento correto durante a execução dos testes.



(a) Falha de restrição na condição de relógio



(b) Falha de relaxamento na condição de relógio

Figura 3. Grids com falhas na condição de relógio

6.4. Falha de reinicialização de relógio

Falhas de reinicialização de relógio são caracterizadas por apresentar transições de um TIOA com o conjunto de reinicialização de relógios distintos do especificado. A con-

sequência são interpretações de relógio alteradas após a execução da transição para valores diferentes daqueles esperados na especificação.

A Figura 4(a) apresenta uma implementação do TIOA da Figura 1(a) com falha de reinicialização, visto que o relógio c_1 não é reinicializado na transição. Num outro cenário, quando assumimos o TIOA da Figura 4(a) como especificação, a Figura 1(a) representa uma implementação com falha de reinicialização ao implementar uma reinicialização inexistente.

Note que mesmo existindo diferenças na estrutura dos autômatos grid das Figuras 1(b) e 4(b), como observado no estado alvo das transições, estes estados representam o mesmo estado do TIOA original. Porém, essas diferenças com transições para estados alvos distintos nos grids não são suficientes para caracterizar um comportamento de forma a detectar essa classe de falhas.

Uma abordagem para detectar essas falhas é observar como as alterações na estrutura do autômato grid influenciam o comportamento do sistema. Quando uma reinicialização especificada deixa de ser implementada, a transição subsequente que possui uma condição para este relógio apresentará falha de restrição da condição de relógio. Como o valor da interpretação de relógio não é reinicializado na transição, os valores anteriormente especificados não poderão ser mais atingidos. De outra forma, quando uma reinicialização especificada não é implementada, a transição subsequente que possui uma condição para este relógio apresentará uma falha de relaxamento em sua condição de relógio. Como o valor da interpretação de relógio na execução da transição é reinicializado, um limite inferior menor que o esperado para as condições de relógio subsequentes pode ser atingido na evolução do sistema.

Apesar de não ser possível caracterizar as falhas de reinicialização de relógio no autômato grid, é possível que a propagação desse tipo de falha seja detectável através da identificação de falhas nas condições de relógio.

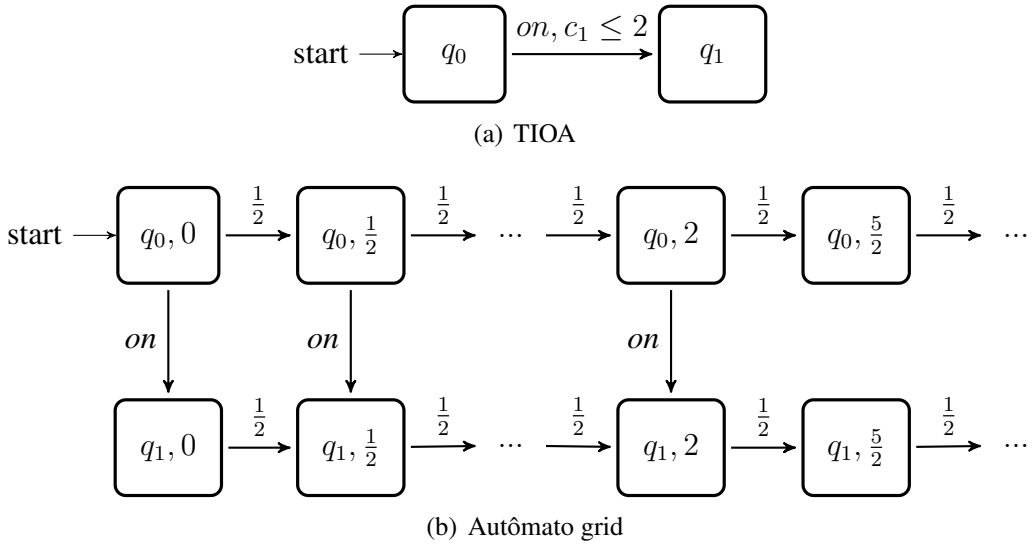
Um terceiro caso particular ocorre quando a transição é implementada com falha de reinicialização de relógio, mas o valor da interpretação do relógio responsável pela falha não é utilizado na condição de relógio de nenhuma transição subsequente. Neste caso, o comportamento do sistema não é alterado, pois independe do relógio ser ou não reinicializado. Essa falha é uma falha considerada não efetiva na execução do sistema.

7. Conclusão

A abordagem de teste baseado em modelos tem sido empregado em sistemas de tempo real. No entanto, sua utilização ainda é um desafio devido a complexidade de se lidar com o aspecto da evolução continua de tempo.

Neste trabalho investigamos a capacidade de se identificar falhas em sistemas de tempo real modelados em TIOA através de autômatos grid. As classes de falha são analisadas de forma que as características de cada tipo de falha possam ser detectadas através da simulação do autômato grid correspondente ao TIOA de especificação. Esta análise foi baseada no modelo de falhas propostos para o modelo TIOA e para o modelo MEF.

Mostramos que as falhas de ação e restrição de relógio podem ser detectadas através de uma cobertura completa das transições do grid.



(a) TIOA

(b) Autômato grid

Figura 4. TIOA sem reinicialização na transição e seu grid correspondente

Já as falhas de transferência podem ser detectadas com a utilização de conjuntos de identificação de estados, pois os modelos grid não possuem a semântica de evolução contínua de tempo. As falhas de relaxamento de condição de relógio apresentam transições extras nos autômatos grid. A detecção dessas transições extras não é possível através da extração de conjuntos de teste a partir de autômatos grid. Uma alternativa é transformar a especificação em modelos completos, de modo que a transição resultante da falha de relaxamento de relógio não seja mais identificada como uma transição extra, mas sim como uma transição com falha de ação.

Observamos também que no caso das falhas de reinicialização de relógio não existe uma forma direta de identificá-las no autômato grid. Contudo, observamos que essa classe de falhas induz a ocorrência de outras falhas no grid. Quando uma operação de reinicialização deixa de ser implementada o autômato grid permite que a falha seja detectada como uma restrição de condição de relógio. Já quando uma operação de reinicialização inexistente é implementada o autômato grid permite que a falha seja detectada como um relaxamento da condição de relógio.

Um direcionamento da pesquisa em andamento é a identificação de outras formas de propagação de falhas que permitam sua detecção assumindo hipóteses e propriedades menos restritivas tanto para a implementação quanto para a especificação. Outro objetivo mais imediato é a formalização das classes de falha apresentadas, de modo a compor precisamente o modelo de falhas para autômatos grid.

Referências

Alur, R. (1999). Timed automata. In *Proceedings of the 11th International Conference on Computer Aided Verification*, CAV '99, pages 8–22, London, UK. Springer-Verlag.

Alur, R. and Dill, D. L. (1994). A theory of timed automata. *Theor. Comput. Sci.*, 126:183–235.

Bonifacio, A. L. and Moura, A. V. (2009). A New Timed Discretization Method for Automatic Test Generation for Timed Systems. Technical Report IC-09-31, Institute

of Computing, University of Campinas. http://www.ic.unicamp.br/∼reltech/2009/09-31.pdf.

Bonifácio, A. L. and Moura, A. V. (2011). A new method for testing timed systems. *Software Testing, Verification and Reliability*, pages n/a–n/a. http://onlinelibrary.wiley.com/doi/10.1002/stvr.454/abstract.

Chow, T. S. (1978). Testing software design modeled by finite-state machines. *IEEE Trans. Softw. Eng.*, 4(3):178–187.

En-Nouaary, A., Dssouli, R., and Khendek, F. (2002). Timed wp-method: Testing real-time systems. *IEEE Trans. Softw. Eng.*, 28:1023–1038.

En-Nouaary, A. and Hamou-Lhadj, A. (2008). A boundary checking technique for testing real-time systems modeled as timed input output automata (short paper). In *Quality Software, 2008. QSIC '08. The Eighth International Conference on*, pages 209 –215.

En-Nouaary, A., Khendek, F., and Dssouli, R. (1999). Fault coverage in testing real-time systems. In *Real-Time Computing Systems and Applications, 1999. RTCSA '99. Sixth International Conference on*, pages 150 –157.

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

Hierons, R. M., Bogdanov, K., Bowen, J. P., Cleaveland, R., Derrick, J., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P., Lüttgen, G., Simons, A. J. H., Vilkomir, S., Woodward, M. R., and Zedan, H. (2009). Using formal specifications to support testing. *ACM Comput. Surv.*, 41(2):1–76.

Kaynar, D. K., Lynch, N. A., Segala, R., and Vaandrager, F. W. (2006). *The Theory of Timed I/O Automata*. Synthesis Lectures on Computer Science. Morgan & Claypool Publishers.

Krichen, M. and Tripakis, S. (2004). Black-box conformance testing for real-time systems. In *In 11th International SPIN Workshop on Model Checking of Software (SPIN'04), volume 2989 of LNCS*, pages 109–126. Springer.

M. Blackburn, R. Busser, A. N. (2004). Why model-based test automation is different and what you should know to get started. In *International Conference on Practical Software Quality and Testing*, Washington, DC, USA. PSQT/PSTT'2004 East.

Springintveld, J., Vaandrager, F., and D'Argenio, P. R. (2001). Testing timed automata. *Theor. Comput. Sci.*, 254:225–257.

Tretmans, J. (2008). Model based testing with labelled transition systems. In Hierons, R. M., Bowen, J. P., and Harman, M., editors, *Formal methods and testing*, pages 1–38. Springer-Verlag, Berlin, Heidelberg.

Utting, M. and Legeard, B. (2007). *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, 1 edition.

Weiglhofer, M. and Wotawa, F. (2009). Improving coverage based test purposes. In *Quality Software, 2009. QSIC '09. 9th International Conference on*, pages 219 –228.

Replicação na Cloud (*of Clouds*)

Alysson Bessani[1]

[1]University of Lisbon, Faculty of Sciences, LaSIGE - Portugal

Abstract. *In this talk we will present some dependability and security aspects of cloud computing, with special emphasis on data and service replication using several cloud providers. We will also describe some experiences in the design and implementation of replicated services in multiple clouds (i.e., the cloud of clouds) together with some related opportunities and challenges for the distributed systems community.*

Resumo. *Nesta palestra serão discutidos alguns aspectos de confiabilidade e segurança no uso da computação em nuvem, com especial atenção ao papel da replicação de dados e serviços usando múltiplos provedores de serviço na nuvem. Serão discutidos as experiências do autor no projeto e implementação de serviços replicados em múltiplas nuvens juntamente com algumas oportunidades e desafios para a comunidade de sistemas distribuídos relacionadas a esse novo cenário.*

Índice por Autor

A

Alchieri, E. A. P. **57**
Arantes, L. **43**

B

Bonifácio, A. L. **131**
Bessani, A. N. **57, 145**

C

Cechin, S. L. **3**
Costa, L. T. **87**
Correa, C. M. **115**

D

da Silveira, M. B. **87**
de Almeida, R. B. **31**
de Santana, A. N. **71**
Doi Júnior, G. **131**
Dotti, F. L. **101, 115**
Duarte Jr., E. P. **43**

F

Fernandes, P. **115**
Fraga, J. da S. **57**

G

Gindri, A. F. **3**
Greve, F. **57, 71**

M

Martins, E. **17**
Maruani, E. **115**
Miranda, J. G. V. **71**

O

Oleksinski, L. G. **115**
Oliveira, F. M. **87**

R

Ribeiro, L. **101**
Rodrigues, E. M. **87**
Rodrigues, L. A. **43**

S

Salas, M. I. P. **17**
Sales, A. **115**
Silva, A. P. C. **31**

V

Vieira, A. B. **31**

W

Weber, T. S. **3**

Z

Zorzo, A. F. **87**