VI Workshop de Testes e Tolerância a Falhas

Fortaleza, 09 de Maio de 2005

Já em sua sexta edição, o WTF é um evento consolidado que promove discussões, impulsiona os avanços e divulga resultados recentes nas áreas de Testes e Tolerância a Falhas. Os trabalhos apresentados são resultados de projetos desenvolvidos principalmente pela academia e indústria brasileiras. Esse ano, o seu comitê de programa foi ampliado para agregar colegas da comunidade lusófona. Pretendemos, dessa maneira, aproximar as nossas comunidades.

Nesta edição, o WTF contemplará atividades de apresentação de trabalhos técnicos na forma de artigos e pôsteres, além de palestra(s) convidada(s). Tal como nos anos anteriores, será realizado em conjunto com o SBRC - Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos, que ocorrerá em Fortaleza de 9 a 13 de maio de 2005. O WTF é promovido pela Sociedade Brasileira de Computação (SBC) e Laboratório Nacional de Redes de Computadores (LARC).

WTF 2005

Organização e Coordenação do Comitê de Programa

- Eliane Martins (IC - UNICAMP)
- Fabíola Greve (DCC - UFBA)

Comitê de Programa

- Avelino F. Zorzo (PUCRS)
- Edmundo Madeira (UNICAMP)
- Eliane Martins (UNICAMP)
- Elias P. Duarte Junior (UFPR)
- Fabíola Greve (UFBA)
- Fernando Pedone (Universita della Svizzera Italiana, SW)
- Francisco V. Brasileiro (UFCG)
- George M. de A. Lima (UFBA)
- Ingrid Jansh-Pôrto (UFRGS)
- Joni S. Fraga (UFSC)
- Jorge Moreira de Souza (Fitec)
- Lau C. Lung (PUCPR)
- Luís Rodrigues (Faculdade de Lisboa, PT)
- Marinho P. Barcellos (UNISINOS)
- Patrícia Machado (UFCG)
- Raul Ceretta (UFSM)
- Rogério De Lemos (University of Kent, UK)
- Rui Oliveira (Universidade do Minho, PT)
- Silvia Vergílio (UFPr)
- Taisy R. Weber (UFRGS)

FreeSkinPreTopic?

Contato:
Profa. Fabíola Greve, E-mail: fabiola@dcc.ufba.br

Endereço para correspondência:
Profa. Fabíola Greve
DCC - Instituto de Matemática.
Av. Adhemar de Barros, S/N, Campus de Ondina. | 40.170-110. Salvador/Bahia.
tel:+55 (71) 3263-6309 | fax:+55 (71) 3263-6276

WTF 2005

Programação do WTF

Nesta edição, o WTF contemplará atividades de apresentação de trabalhos técnicos na forma de artigos e resumos, além de palestras convidadas. Os artigos, escolhidos pela sua qualidade e organizados em cinco sessões técnicas, denunciam temas de pesquisa atuais e concentram-se nas áreas de roteamento e tolerância a falhas em ambientes móveis e entre pares, protocolos adaptativos de consenso e detecção de falhas, além de testes e injeção de falhas. Ao final de cada sessão será feita uma rodada de discussões envolvendo os autores dos respectivos artigos.

Para enriquecer e fomentar mais ainda as discussões durante o evento, convidamos palestrantes especialistas nas respectivas áreas dos artigos apresentados nas sessões. Professor Henrique Madeira, da Universidade de Coimbra, nos falará sobre injeção de falhas de software realistas e perspectivas de utilização para ``risk assessment''. Professor Francisco Brasileiro, da UFCG, fará uma apresentação sobre protocolos de consenso adaptativos. Professor Edmundo Madeira, da UNICAMP, abordará aspectos de qualidade de serviço em detecção de falhas.

8:30 a 8:50 - Abertura do WTF

Sessão Técnica 1: Tolerância a Falhas em Ambientes Móveis e entre Pares

Chair	Raul Ceretta Nunes
[8:50-9:10]	Coordenação Desacoplada Tolerante a Faltas Bizantinas
[9:10-9:30]	Avaliando Aspectos de Tolerância a Falhas em Protocolos de Roteamento para Redes de Sensores sem Fio
[9:30-9:50]	Roteamento Tolerante a Falhas Baseado em Caminhos Robustos
[9:50-10:10]	Avaliação de um Mecanismo de Checkpointing para o MyGrid?
[10:10-10:30]	Discussões
[10:30-11:00]	INTERVALO

Sessão Técnica 2: Testes e Injeção de Falhas

Chair	Eliane Martins
[11:00-11:40]	**Palestra Convidada**: Injection of Realistic Software Faults: Experimental Risk Assessment
	Professor Henrique Madeira (Univ. Coimbra)
[11:40-12:00]	Um Injetor de Falhas de Comunicação construído usando Programação Orientada Aspectos
[12:00-12:20]	Xception Fault Injection and Robustness Testing Framework: a Case-Study of Testing RTEMS
[12:20-12:30]	Discussões
[12:30-	INTERVALO

14:00]

Sessão Técnica 3: Detecção de Falhas

Chair	Elias Procópio Duarte Jr.
[14:00-14:40]	**Palestra convidada**: Qualidade de Serviço em Detectores de Falhas na Presença de Rajadas de Perdas de Pacotes
	Professor Edmundo Madeira (UNICAMP)
[14:40-15:00]	Engineering a Failure Detection Service for Widely Distributed Systems
[15:00-15:10]	Discussões

Sessão Técnica 4: Testes e Avaliação de Desempenho

Chair	Marinho Barcellos
[15:10-15:20]	Estendendo FIONA para uma arquitetura híbrida
[15:20-15:30]	Testando uma Infra-estrutura FT-CORBA: O Caso GroupPac?
[15:30-15:40]	Comunicação Fim-a-Fim a Alta Velocidade em Redes Gigabit
[15:40-16:00]	Discussões
[16:00-16:30]	INTERVALO

Sessão Técnica 5: Protocolos de Consenso

Chair	Ingrid Jansch-Porto
[16:30-17:10]	**Palestra convidada**: Consenso Indulgente Adaptativo
	Professor Francisco Brasileiro (UFCG)
[17:10-17:30]	Fast Adaptable Uniform Consensus Using Global State Digests
[17:30-17:50]	Integração das Especificações ROMIOP e ETF para Difusão Atômica no CORBA
[17:50-18:00]	Discussões

18:00-19:00 - Encerramento do WTF e Reunião do Comitê

WTF 2005

Palestrantes Convidados

Injection of Realistic Software Faults: Experimental Risk Assessment, *Professor Henrique Madeira*

Departamento de Engenharia Informática, Universidade de Coimbra, Portugal

Resumo This talk presents a set of operators for the injection of realistic software faults through low-level code mutations. The definition of these operators was based on the analysis of an extensive collection of real software faults obtained from several programs, including user applications and operating systems code. Using Orthogonal Defect Classification (ODC) from IBM as a starting point, faults were classified in a detailed manner according to the high-level constructs where the faults reside and their effects in the program. We observed that a large percentage of faults fall in well-defined classes and can be characterized in a very precise way, allowing accurate emulation through a small set of well-defined mutation operators. The resulting operators closely emulate a broad range of common programmer mistakes. Furthermore, as the mutation is performed directly at the executable code, software faults can be injected in targets for which the source code is not available. This allows the injection of realistic software faults in a broad range of application areas and purposes. In the talk, we will mainly discuss the use of this new approach for experimental risk assessment in COTS based software development.

Qualidade de Serviço em Detectores de Falhas na Presença de Rajadas de Perdas de Pacotes, *Professor Edmundo Madeira*

Instituto de Computação, UNICAMP, Brasil

Resumo: A Qualidade de Serviço (QoS?) de detectores de falhas determina a rapidez com que um detector de falhas detecta a quebra de um processo, e a precisão com que o detector informa essa quebra. Em Redes de Longa Distância e em Redes sem Fio, a ocorrência de quebra de processos, variações de atraso grandes e rajadas de perdas são freqüentes. Esta palestra aborda a oferta de QoS? por parte dos detectores de falhas. Inicialmente, serão analisados as métricas de QoS?, os algoritmos e os configuradores propostos por Chen, Toueg e Aguilera, e alguns outros trabalhos sobre QoS? em sistemas distribuídos. Em seguida, será detalhado um configurador de detector de falhas que considera a distribuição probabilística de comprimentos de rajadas de perdas de pacotes usando um modelo de Markov.

Consenso Indulgente Adaptativo, *Professor Francisco Brasileiro*

Departamento de Sistemas e Computação, Universidade Federal de Campina Grande, Brasil

Resumo: O consenso constitui uma importante abstração para a construção de sistemas distribuídos tolerantes a faltas, sendo utilizado como um bloco básico para a implementação de soluções de problemas de acordo, tais como confirmação atômica, difusão atômica e filiação de grupo. Normalmente, o consenso é implementado como um módulo independente, com interface bem definida, o qual é usado pelos serviços distribuídos de acordo. Um dos atrativos para implementar serviços baseados em consenso está na facilidade de validar (provar correção) tais serviços quando comparados àqueles não baseados em consenso. Por outro lado, soluções baseadas em consenso, normalmente, geram uma grande carga no sistema. Sendo assim, questões de desempenho do consenso têm sido amplamente discutido na literatura. Em particular, existem vários trabalhos sobre questões de desempenho do consenso indulgente.

Muitos protocolos de consenso indulgente são assimétricos no que diz respeito aos papéis assumidos por seus participantes. Nesse caso, um dos processos participantes desempenha o papel de coordenador, enquanto os demais cooperam com o coordenador para realizarem uma determinada tarefa. No sentido de tolerar faltas do processo coordenador, protocolos assimétricos dependem de ordenação consistente de processos. A escolha dos participantes que desempenham o papel de coordenador é feita através de um lista ordenada de processos; quando o coordenador corrente falha, o próximo processo da lista é selecionado para assumir este papel. Normalmente, o mecanismo utilizado para ordenação de processos é bastante simples. Este mecanismo consiste em ordenar a lista a priori, i.e., antes da execução do protocolo, com base nos identificadores dos processos. O problema desta solução é que a ordenação dos processos não considera as condições do ambiente ao longo da execução do protocolo. Sendo assim, o protocolo pode ter perdas de desempenho, em cenários de carga heterogênea e dinâmica (característica dos sistemas distribuídos reais), quando a ordenação definida a priori torna-se inadequada.

Adaptação é um requisito fundamental para lidar com questões de desempenho em sistemas heterogêneos e dinâmicos. Sendo assim, é possível melhorar o desempenho de protocolos de consenso indulgentes que dependem de ordenação de processos através do uso de mecanismos adequados de adaptação. Por exemplo, protocolos de consenso adaptativos podem ser construídos seguindo uma abordagem adaptativa para ordenação de processos. Tal abordagem consiste em usar informações sobre a situação de carga do ambiente de execução para ordenar a lista de processos requerida pelos protocolos de consenso. Nesta palestra iremos discutir o projeto e a implementação desse tipo de protocolo.

Coordenação Desacoplada Tolerante a Faltas Bizantinas

Alysson Neves Bessani[1*], **Joni da Silva Fraga**[1†], **Lau Cheuk Lung**[‡2]

[1]DAS - Departamento de Automação e Sistemas
UFSC - Universidade Federal de Santa Catarina

[2]PPGIA - Programa de Pós-Graduação em Informática Aplicada
PUC-PR - Pontifícia Universidade Católica do Paraná

{neves,fraga}@das.ufsc.br, lau@ppgia.pucpr.br

Abstract. *Existing distributed systems require communication mechanisms that satisfy requirements as anonymity and temporary disconnection. In this context, generative communication is becoming one of the coordination models capable to address these requirements once it is decoupled in time and space. This work presents the first proposal in the literature to consider the development of a byzantine fault tolerant generative coordination infrastructure. This construction is based on the byzantine quorum systems replication technique.*

Resumo. *Os sistemas distribuídos atuais requerem mecanismos de comunicação que atendam requisitos como anonimato e desconexão temporária. Neste contexto, a comunicação generativa vêm se afirmando como um dos modelos de coordenação capazes de atender esses requisitos uma vez que é desacoplada no tempo e no espaço. Este trabalho apresenta a primeira proposta da literatura a considerar a construção de uma infra-estrutura de coordenação generativa tolerante a faltas bizantinos. Esta construção se dá através da aplicação de replicação por sistemas de quóruns bizantinos.*

1. Introdução

O crescimento do uso dos computadores e da Internet em todas as áreas de conhecimento estimula o desenvolvimento de novas tecnologias e aplicações distribuídas. Os participantes destas novas aplicações são caracterizados pela heterogeneidade, capacidades de comunicação variável e por serem pouco confiáveis. Em um futuro próximo, a necessidade de sistemas para suporte a aplicações distribuídas em larga escala se tornará comum, e para tratar deste potencial problema, novos modelos e paradigmas de computação (p.ex. redes par a par, serviços web e computação em grade) estão sendo propostos. Em muitos aspectos, estes modelos/paradigmas se baseiam no conceito de sistemas abertos, que se caracterizam principalmente por serem compostos por um número desconhecido de processos heterogêneos que participam das interações em diferentes momentos.

Os mecanismos de coordenação [Gelernter and Carriero, 1992] usualmente empregados para a interação em sistemas distribuídos, como a comunicação por passagem de mensagens, não são adequados para os sistemas abertos do futuro. O requisito de anonimato e as

[*]Doutorando Bolsista PGI/CNPq.
[†]Bolsista PQ/CNPq.
[‡]Bolsista PQ/CNPq.

comunicações pouco confiáveis implicam diretamente na necessidade de interações desacopladas nestes sistemas. Assim, modelos de coordenação alternativos se fazem necessários. Dentre estes modelos, a comunicação generativa [Gelernter, 1985] se destaca pela sua flexibilidade e simplicidade, sendo utilizada em uma série de infra-estruturas de comunicação como o JAVASPACES [Sun Microsystems, 2003] e o TSPACES [Lehman et al., 2001]. Neste modelo, os processos interagem através de um espaço de memória compartilhado (espaço de tuplas) em que estruturas de dados genéricas (tuplas) são colocadas, lidas e coletadas durante as interações. A coordenação ocorre de maneira desacoplada no tempo (os participantes não precisam estar engajados ao mesmo tempo) e no espaço (não precisam conhecer uns aos outros) [Cabri et al., 2000]. Além disso, o número reduzido de operadores e sua generalidade implicam em uma grande simplicidade (em termos de programação) na implementação de sistemas distribuídos [Carriero and Gelernter, 1989, Sun Microsystems, 2003].

Independentemente do modelo de coordenação empregado nos sistemas distribuídos abertos, estas interações estão sujeitas a todos os tipos de falhas e ataques de segurança. Estes eventos podem ser agrupados e modelados como faltas bizantinas [Lamport et al., 1982], para que técnicas de tolerância a faltas possam ser empregadas visando melhorar a confiabilidade da infra-estrutura da coordenação. A aplicação destas técnicas permite que uma infra-estrutura de coordenação para aplicações críticas justifique a confiança depositada neste componente fundamental em sistemas abertos.

Este trabalho apresenta a primeira proposta de modelo de coordenação baseado em espaço de tuplas tolerante a faltas bizantinas. Esta proposta se baseia na emulação do espaço de tuplas sobre um sistema distribuído, sem memória compartilhada, onde os processos se comunicam por passagem de mensagens. A técnica utilizada para esta emulação são os sistemas de quóruns bizantinos [Malkhi and Reiter, 1998, Martin et al., 2001].

O desenvolvimento desta infra-estrutura de coordenação objetiva atacar o problema da tolerância a intrusões [Fraga and Powell, 1985, Veríssimo et al., 2003] em sistemas abertos. Para tanto, o ambiente considerado é bastante desfavorável: número desconhecido de processos sujeitos a faltas maliciosas[1] executando em um ambiente completamente assíncrono.

Este artigo está estruturado da seguinte forma: a seção 2 apresenta uma breve introdução ao modelo de coordenação generativa. A seção 3 apresenta as premissas de nosso modelo de sistema, o tipo de sistema de quóruns utilizado e os protocolos que implementam o espaço de tuplas tolerante a faltas bizantinas. A seção 4 apresenta um estudo a respeito da complexidade e dos custos dos protocolos desenvolvidos. Finalmente, as seções 5 e 6 apresentam alguns trabalhos relacionados presentes na literatura e as considerações finais deste artigo, respectivamente.

2. Coordenação Generativa

O modelo de **coordenação generativa** foi introduzida no contexto da linguagem de programação para sistemas paralelos LINDA [Gelernter, 1985]. O mecanismo de coordenação definido nesta linguagem permite aos processos distribuídos interagirem sobre um espaço de memória compartilhado em que estruturas de dados genéricas (tuplas) são adicionadas, lidas e removidas. A comunicação é chamada generativa devido ao fato de que uma vez criadas no espaço, as tuplas têm existência completamente separada dos processos

[1] Processos que utilizam o espaço de tuplas para coordenação.

que as geraram.

No modelo generativo, uma tupla $t = \langle f_1, f_2, ..., f_m \rangle$ é um conjunto de campos ordenados onde cada campo f_i está associado a um tipo (inteiro, lógico, *string*, etc...) e pode ter um valor definido que deve estar no domínio deste tipo. Uma tupla em que todos os campos têm valores definidos (campos atuais) é chamada **entrada** (*Entry*). Um **molde** (*template*), denotado por $\bar{t}$, é uma tupla que pode conter campos sem valor definido (campo formal).

O **tipo de um campo**, seja ele atual ou formal, é dado pela função $\tau : \mathcal{F} \rightarrow \mathcal{T}$, onde $\mathcal{F}$ é o conjunto de todos os possíveis campos, atuais ou formais, e $\mathcal{T}$ é o conjunto dos possíveis tipos de dados assumidos no modelo de computação usado. O **tipo de uma tupla** t, denotado por $type(t)$, é a sequência dos tipos dos campos de t. Diz-se que duas tuplas t e t' são do mesmo tipo ($type(t) = type(t')$) se os campos destas tupla são do mesmo tipo. Em princípio, duas tuplas combinam se elas são do mesmo tipo e têm valores compatíveis.

As manipulações realizadas no espaço de tuplas consistem de invocações de três operações básicas [Gelernter, 1985][2]:

- $out(t)$: Esta operação adiciona a tupla t no espaço de tuplas. A operação out é chamada usualmente de operação de escrita no espaço;
- $in(\bar{t})$: Esta operação retira do espaço de tuplas uma tupla que combine com o molde $\bar{t}$. Caso não exista nenhuma tupla que combine com $\bar{t}$ no espaço o processo fica parado esperando até que exista uma (operação bloqueante). A operação in é usualmente chamada de leitura destrutiva, remoção ou coleta;
- $rd(\bar{t})$: Operação que lê no espaço de tuplas uma tupla que combine com o molde $\bar{t}$. Caso não exista nenhuma tupla que combine com $\bar{t}$ no espaço, o processo fica parado esperando até que exista uma (operação bloqueante). A grande diferença desta operação para in é o fato dela não remover a tupla lida do espaço de tuplas.

São também definidas duas variantes das operações in e rd não bloqueantes, estas operações são chamadas inp e rdp e seus funcionamentos são exatamente iguais ao das operações originais, a não ser pelo fato de que elas sempre retornam um valor lógico: se não houver uma tupla no espaço que combine com o molde passado, estas operações retornam um valor de falha $false$, caso contrário, um valor $true$ é retornado juntamente com a tupla lida.

Uma característica muito importante do modelo generativo e das operações definidas para o espaço de tuplas é seu não determinismo inerente. Se várias tuplas existentes no espaço combinam com um molde passado como parâmetro em uma operação in ou rd, qualquer uma delas pode ser retornada. Da mesma forma, se dois ou mais processos estiverem parados esperando por tuplas com determinadas características (definidas nos moldes apresentados como argumentos nas operações) e uma entrada é inserida no espaço de tal forma que ela combina com os moldes de qualquer um dos processos esperando a tupla, qualquer um deles pode recebê-la, permanecendo os demais em estado de espera.

Outra característica fundamental da comunicação generativa é o acesso associativo a tuplas: os dados são acessados a partir de seu conteúdo, e não através de seu endereço. A idéia é que o espaço de tuplas é uma sacola onde ítens de dados (tuplas) são colocados, lidos e retirados de acordo com suas características (conteúdo). Estes dados, uma vez no espaço de

[2]Existem diversas definições da semântica de um espaço de tuplas, todas ligeiramente diferentes. Neste trabalho estamos assumindo a definição original adaptada para sistemas utilizados atualmente como o JAVASPACES [Sun Microsystems, 2003] e o TSPACES [Lehman et al., 2001].

tuplas, não podem ser alterados. Desta forma, para se alterar uma tupla do espaço, é preciso removê-la, e criar outra tupla no espaço com os valores da antiga, porém alterada. Esta característica de memória associativa diferencia este modelo de coordenação dos demais modelos baseados em memória compartilhada. A associatividade da memória, implementada através de um esquema de nomeação estruturado [Gelernter, 1985] parecido com a operação *select* dos bancos de dados relacionais, permite que as operações in e rd acessem as tuplas do espaço através de seus valores e não de seus endereços. Fundamental para este mecanismo é o conceito de **combinação de tuplas**. Diz-se que duas tuplas (com o mesmo número de campos), uma entrada $t = \langle f_1, f_2, ..., f_m \rangle$ e um molde $\overline{t} = \langle \overline{f}_1, \overline{f}_2, ..., \overline{f}_m \rangle$, combinam[3] denotada por $m(t, \overline{t})$, se e somente se $\forall i = 1..m : \tau(f_i) = \tau(\overline{f}_i) \wedge (\overline{f}_i = \perp \vee f_i = \overline{f}_i)$. Esta condição define que a combinação existe se para todo o campo da tupla, o campo correspondente do molde é do mesmo tipo e tem o mesmo valor ou tem valor indefinido ($\perp$). A definição de combinação de tupla implica diretamente no fato de que $\forall t, \overline{t} : m(t, \overline{t}) \rightarrow (type(t) = type(\overline{t}))$.

3. Espaço de Tuplas Tolerante a Faltas Bizantinas

O ponto central do trabalho aqui apresentado é a provisão de um espaço de tuplas tolerante a faltas bizantinas. A replicação é a técnica chave nesse sentido, e a coordenação das réplicas do espaço (para manutenção de sua consistência) é o principal problema a ser resolvido.

3.1. Modelo de Sistema

O modelo de sistema adotado consiste de um conjunto arbitrário de processos clientes que se comunicam com servidores que emulam um espaço de tuplas. Todas as computações e comunicações ocorrem de forma **assíncrona** [Fischer et al., 1985]: não existem limites de tempo para sua terminação.

Em termos de falhas de processos, assumimos que um número arbitrário de clientes e um limite máximo de f servidores estão sujeitos a **falhas bizantinas**: podem desviar arbitrariamente de sua especificação e podem, inclusive, trabalhar em conluios maliciosos visando corromper o sistema. Assumimos ainda **independência de faltas**, obtida através da utilização de diferentes plataformas (hardware, SO, VM, etc) [Castro et al., 2003].

Todos os processos do sistema (clientes ou servidores) se comunicam através de canais ponto a ponto confiáveis com ordenação FIFO. Assume-se também que cada processo tem um identificador único e que existe um esquema de assinaturas digitais não forjáveis [Rivest et al., 1978] que permite que todas as mensagens trocadas nos protocolos sejam autenticadas.

O espaço de tuplas emulado sobre um sistemas de quóruns bizantinos torna disponíveis as operações definidas na seção 2, porém estas não são executadas de forma atômica (indivisível) devido ao não determinismo das réplicas e as características do ambiente assíncrono. Assim, toda operação o tem um início e um fim (denotados $begin(o)$ e $end(o)$), que são os eventos que marcam a invocação da operação o e a definição de sua resposta no cliente, respectivamente. É possível montar um ordem total destes eventos em um sistema se tomarmos os instantes de tempo real que eles são executados. A esta sequência de inícios e fins de operações damos o nome de **execução**.

[3] A regra de combinação apresentada neste texto são adaptações livres das apresentadas em [Busi et al., 2003], visando torná-las compatíveis com a literatura corrente.

A partir desta ordem total, podemos definir, para quaisquer dois eventos e_1 e e_2, qual acontece antes de outro (denotado por $\rightarrow$). Por exemplo, para toda operação o, $begin(o) \rightarrow end(o)$. Dadas duas operações o_1 e o_2, dizemos que o_1 **precede** o_2 se $end(o_1) \rightarrow begin(o_2)$. Se $end(o_1) \nrightarrow begin(o_2)$ e $end(o_2) \nrightarrow begin(o_1)$, então o_1 e o_2 são ditas **concorrentes**. Para simplificar os algoritmos, assumimos que duas operações executadas por um mesmo processo **nunca** são concorrentes.

3.2. Sistemas de Quóruns Bizantinos

Os **sistemas de quóruns bizantinos** [Malkhi and Reiter, 1998] são uma alternativa para a emulação de objetos de memória compartilhada em sistemas distribuídos onde os processos se comunicam por passagem de mensagens. O grande atrativo desta abordagem é o fato de que seus algoritmos não requerem a resolução do problema de consenso, não estando portanto sujeitos a impossibilidade FLP [Fischer et al., 1985].

Um sistema de quóruns para um universo de servidores de dados é um conjunto de vários sub-conjuntos de servidores, chamados quóruns, que possuem intersecção entre si. O princípio por trás de seu uso em serviços de armazenamento é que, se uma variável compartilhada é replicada entre todos esses servidores, as operações de leitura e escrita precisam ser feitas apenas em um dos quóruns destes servidores, e não em todo o sistema. A existência de intersecções entre os quóruns permite a construção de protocolos de leitura e de escrita que mantém a integridade da variável compartilhada mesmo que estas operações sejam realizadas em diferentes quóruns.

Formalmente, considera-se um universo U (finito) de n servidores ($|U| = n$) e um conjunto arbitrário de clientes Π. O sistema de quóruns bizantinos é um conjunto de sub-conjuntos de servidores (quóruns) $\mathcal{Q} \subseteq 2^U$ onde todos os quóruns $Q \in \mathcal{Q}$ têm em suas intersecção um número suficiente de servidores (propriedade de consistência) e sempre existe pelo menos um quórum onde não existem servidores faltosos (propriedade de disponibilidade) [Malkhi and Reiter, 1998].

Os servidores implementam objetos de memória compartilhada sendo organizados como um sistema de quóruns de mascaramento que tolera até f faltas [Malkhi and Reiter, 1998]. Em particular, utilizamos um **sistema de quóruns de mascaramento assimétrico** [Martin et al., 2001], que se distingue dos demais devido ao fato de usar diferentes tamanhos de quóruns em diferentes operações.

Devido as exigências do sistemas de quóruns assimétricos, assumimos que $n \geq 3f + 1$ e que os dois tipos de quóruns presentes no sistema, de leitura (Q_r) e escrita (Q_w), contém $\lceil \frac{n+f+1}{2} \rceil$ e $\lceil \frac{n+f+1}{2} \rceil + f$ servidores, respectivamente. A figura 1 apresenta um exemplo de sistema de quóruns com $n = 4$ e $f = 1$.

Nesta figura temos representados o quórum de escrita $Q_w = \{s_1, s_2, s_3, s_4\}$ e um quórum de leitura[4] $Q_r = \{s_2, s_3, s_4\}$. Dois clientes, c_1 e c_2, interagem com o sistema de quóruns chamando operações de escrita e leitura nos quóruns Q_w e Q_r, respectivamente.

3.3. Protocolo

Nesta seção apresentamos os algoritmos para as operações definidas na comunicação generativa considerando um espaço de tuplas replicado em um sistema de quóruns bizantinos. Todos os algoritmos definidos consideram que cada servidor s do sistema tem uma cópia do espaço de tuplas, chamado espaço local e denotado por T_s.

[4]Todo conjunto de 3 servidores neste sistema é um quórum de leitura neste sistema.

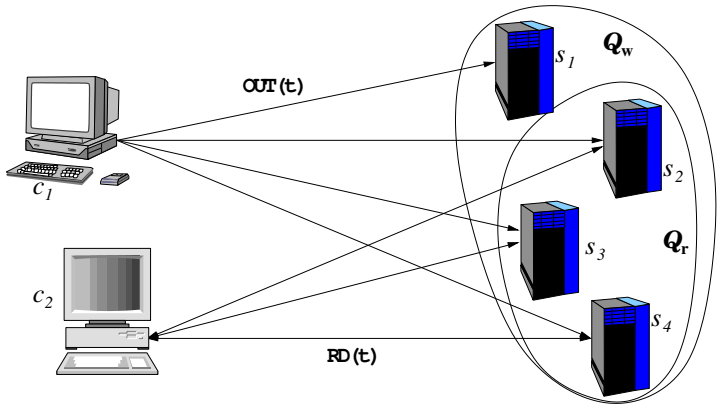


Figura 1: Sistema de quóruns mascaramento assimétrico ($n = 4$ e $f = 1$).

A definição formal das condições de correção e vivacidade destes protocolos bem como a prova de que os mesmos satisfazem essas condições pode ser encontrada na versão estendida desse artigo [Bessani et al., 2005a].

3.3.1. Operação out

A inclusão de uma tupla t no espaço de tuplas replicado é realizada através da invocação da operação out. Esta operação é implementada pelo algoritmo 1.

Algoritmo 1 Operação out (cliente c e servidor s).

1: {**Parte Cliente**}	1: {**Parte Servidor**}
2: **procedure** $out(t)$	**Require:** $receive(c, \langle \text{OUT}, t \rangle)$
3: $\forall s \in Q_w, send(s, \langle \text{OUT}, t \rangle)$	2: $T_s \leftarrow T_s \cup \{t\}$

Devido aos requisitos fracos em termos de sincronismo do modelo generativo e pela premissa de canais confiáveis o cliente não necessita de confirmações de que sua tupla foi recebida pelos servidores. Este tipo de protocolo de escrita, chamado **sem confirmação** [Martin et al., 2002], é mais leve e pode ser usado em sistemas onde o escritor não necessita ter certeza do momento em que sua escrita termina. No caso do algoritmo 1, a escrita termina quando um quórum de leitura completo de servidores corretos recebe a tupla, pois desta forma a tupla inserida poderá ser lida.

3.3.2. Operação rdp

A leitura não destrutiva de uma tupla t, que combina com um molde $\bar{t}$ passado pelo cliente c na invocação de uma operação $rdp(\bar{t})$, acontece através do algoritmo 2.

O funcionamento do algoritmo no cliente consiste basicamente em uma pesquisa nos servidores do sistema tentando obter as tuplas que casam com o molde $\bar{t}$ (linha 3). Cada servidor s acessado responde com um conjunto contendo suas tuplas que combinam com $\bar{t}$ ($T_s^{\bar{t}}$), conforme apresentado nas linhas 2 e 3 do algoritmo servidor. As respostas são coletadas[5] (laço da linha 4-10) até que o cliente consiga definir uma tupla que aparece em pelo menos

[5]O cliente considera apenas uma resposta por servidor.

Algoritmo 2 Operação rdp (cliente c e servidor s).

```
 1: {Parte Cliente}
 2: procedure rdp(t̄)
 3: ∀s ∈ U, send(s, ⟨RD, t̄⟩)
 4: repeat
 5:    wait until receive(s, ⟨RSP-RD, T_s^t̄⟩))
 6:    X ← X ∪ {T_s^t̄}
 7:    if ∃t : (|{T ∈ X : t ∈ T}| = |Q_r|) then
 8:       return t
 9:    end if
10: until |{T ∈ X : T = ∅}| = |Q_r| − f
11: return ⊥
```

```
 1: {Parte Servidor}
Require: receive(c, ⟨RD, t̄⟩)
 2: T_s^t̄ ← {t ∈ T_s : m(t, t̄)}
 3: send(c, ⟨RSP-RD, T_s^t̄⟩)
```

um quórum de leitura completo, sendo esta tupla o retorno da operação (linha 8), ou $|Q_r| - f$ processos retornem um conjunto vazio, indicando não haver tal tupla (linhas 10 e 11).

3.3.3. Operação inp

A operação inp é a que exige o protocolo mais complexo para sua implementação sobre sistemas de quóruns bizantinos. Esta complexidade se deve basicamente ao fato de uma mesma tupla nunca poder ser coletada do espaço de tuplas por duas operações inp. Este requisito implica em uma exclusão mútua natural entre clientes que tentam remover um mesmo tipo de tupla concorrentemente.

Nossa proposta de implementação para esta operação utiliza o algoritmo de exclusão mútua **Confeiteiro Bizantino** [Bessani et al., 2005b], implementado sobre o mesmo sistema de quóruns bizantinos em que estão localizadas as réplicas do espaço de tuplas. Este algoritmo requer também $n \geq 3f + 1$ e está definido sobre duas primitivas: $enter(r)$, que tenta obter acesso ao recurso r, e $exit(r)$, que libera o acesso ao recurso r. O algoritmo do confeiteiro bizantino satisfaz as seguintes propriedades [Lynch, 1996, Bessani et al., 2005b]:

Exclusão Mútua: Não existe um estado global do sistema onde dois processos corretos estão em seção crítica (executaram $enter(r)$ e não começaram a executar $exit(r)$);

Progresso justo (*Starvation-freedom*): Se um processo correto está na região de entrada (executando $enter(r)$), então **este** processo acabará por executar sua região crítica (terminando a sua execução de $enter(r)$).

Utilizando um algoritmo com estas propriedades, torna-se trivial a implementação da operação $inp(\bar{t})$: o processo tenta obter acesso ao tipo da tupla que deseja obter e quando consegue, lê a tupla (através de $rdp(\bar{t})$) e então envia uma mensagem aos servidores removendo a tupla. Ao receber a confirmação de um quórum completo de que a tupla foi apagada, ele libera o tipo da tupla. O algoritmo 3 apresenta esse protocolo.

Durante a execução do algoritmo 3 o cliente, após obter o acesso ao tipo do molde desejado (linha 3) e executar uma leitura com esse molde (linha 4), verifica se existe uma tupla que pode ser removida (o resultado da leitura deve ser diferente de $\perp$ - linha 5). Em caso afirmativo, uma requisição de remoção é enviada a um quórum de escrita de servidores (linha 6) e respostas são coletadas até que $|Q_r| - f$ servidores confirmem a remoção da tupla (linha 14) ou um quórum de leitura negue a remoção (linha 10). O quórum de $|Q_r| - f$ servidores corresponde exatamente a quantidade de servidores que removendo a tupla de

Algoritmo 3 Operação inp (cliente c e servidor s).

```
 1: {Parte Cliente}
 2: procedure inp(t̄)
 3: enter(type(t̄))
 4: t ← rdp(t̄)
 5: if t ≠ ⊥ then
 6:    ∀s ∈ Q_w, send(s, ⟨IN, t⟩)
 7:    repeat
 8:       wait until receive(s, ⟨RSP-IN, r⟩)
 9:       R ← R ∪ {⟨s, r⟩}
10:       if |{⟨s, r⟩ ∈ R : r = ⊥}| = |Q_r| then
11:          exit(type(t̄))
12:          re-begin procedure
13:       end if
14:    until |{⟨s, r⟩ ∈ R : r = t}| = |Q_r| − f
15: end if
16: exit(type(t̄))
17: return t
```

```
 1: {Parte Servidor}
Require: receive(c, ⟨IN, t⟩)
 2: if lockedFor(type(t), c) then
 3:    T_s ← T_s \ {t}
 4:    send(c, ⟨RSP-IN, t⟩)
 5: end if
Require: c ∈ susp_◇M ∧ lockedFor(type(t), c)
 6: unlock(type(t))
 7: send(c, ⟨RSP-IN, ⊥⟩)
```

seus espaços locais a tornam inexistente no espaço global, não podendo portanto ser lida ou removida em futuras operações.

As linhas 2-5 da parte do servidor do algoritmo 3 correspondem a resposta do servidor para a requisição de remoção. Note que um servidor só responde a uma requisição deste tipo se o tipo da tupla requisitada está "travado" para o processo requisitante. Como um cliente faltoso pode obter acesso a um tipo de tupla e não liberá-lo (devido ao seu caráter malicioso ou a uma eventual falta de parada), impedindo que outros processos possam remover tuplas deste tipo, faz-se necessário um mecanismo adicional que permita aos servidores manter a acessibilidade do espaço. O mecanismo mais adequado para este fim é a introdução de premissas de sincronismo fraco encapsuladas em um detector de falhas da classe $\diamondsuit\mathcal{PM}$ [Bessani et al., 2005b], uma classe de detectores de mudez [Doudou et al., 1999] "equivalente" ao detector de faltas de parada $\diamondsuit\mathcal{P}$ [Chandra and Toueg, 1996]. Assim, cada servidor do sistema utiliza um detector desta classe para monitorar seus clientes. Se um cliente é suspeito pelo detector de um servidor enquanto está de posse do tipo da tupla o servidor revoga esta posse ("destravando" sua cópia local) e envia uma notificação ao cliente contendo $r = \perp$ (linhas 6 e 7 da parte servidor).

A utilização do detector de falha pelos servidores se reflete no use de *timeouts* na implementação do algoritmo. Desta forma, o cliente que obtém acesso a um tipo de tupla tem um limite de tempo finito para realizar a remoção da tupla escolhida. Caso este cliente (faltoso ou não) não consiga realizar esta remoção, ele volta ao início e tenta re-executar todo o algoritmo novamente (linhas 11 e 12). Note que o detector de mudez é usado apenas para garantir que todo processo que obtém acesso a um tipo de tupla termine por liberá-lo, garantindo assim a vivacidade do algoritmo.

Utilizando o algoritmo do confeiteiro bizantino para exclusão mútua, o algoritmo 3 requer 9 passos de comunicação[6] para terminar quando executado sem concorrência e 11 quando houver mais de um processo tentando obter acesso ao mesmo tipo de tupla. Este número, inaceitável na prática, pode ser diminuído para 6 (sem concorrência) e 8 passos

[6]Métrica que considera o número de comunicações sequenciais entre processos.

de comunicação fazendo-se com que a leitura seja feita durante a execução do algoritmo de exclusão mútua (utilizando-se as mesmas mensagens) e a mensagem de saída da região crítica seja enviada juntamente com a mensagem a notificação de remoção (linha 6). A figura 2 ilustra a execução deste protocolo sem e com a otimização[7]. As trocas de mensagens do confeiteiro bizantino são descritas em [Bessani et al., 2005b].

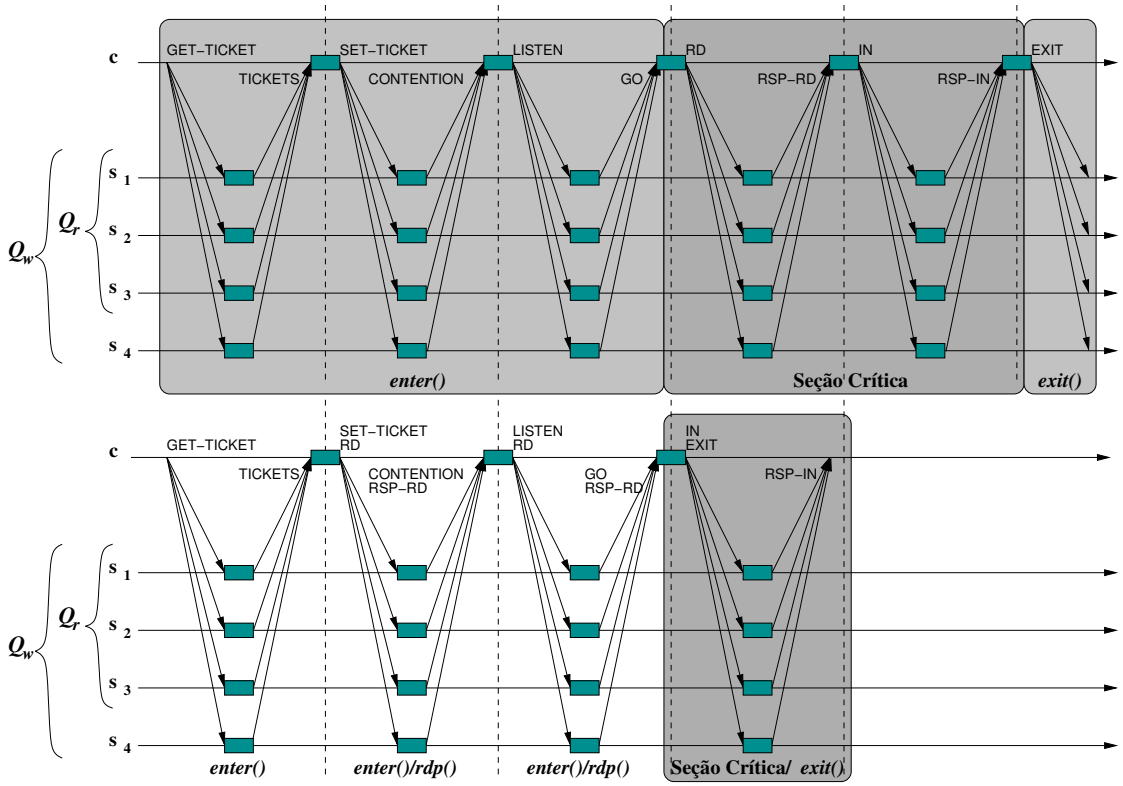


Figura 2: Execuções da operação inp sem e com otimização ($n = 4$ e $f = 1$).

3.3.4. Operações Bloqueantes

As operações bloqueantes rd e in podem ser implementadas fazendo-se com que o cliente execute rdp e inp repetidamente até conseguir a tupla [Xu and Liskov, 1989]. Apesar desta estratégia não ser ótima em termos de desempenho, ela é válida e está de acordo com o não determinismo inerente do espaço de tuplas [Gelernter, 1985].

4. Análise dos Protocolos

A tabela 1 apresenta o custo dos protocolos apresentados. As métricas consideradas são: tamanho das mensagens (T.M.), complexidade de mensagens (C.M.) e passos de comunicação (P.C.).

Os valores da tabela 1 consideram apenas execuções em que os detectores de mudez dos servidores não suspeitam de nenhum processo correto. As operação out e rdp são bastante simples, e portanto têm um custo bastante reduzido se comparadas à operação inp. Esta operação tem seu custo domindo pelo algoritmo de exclusão mútua e portanto exige vários

[7]O padrão de comunicação apresentado na figura 2 não leva em consideração o fato do sistema ser assíncrono. Esta simplificação foi feita visando melhorar a legibilidade da figura.

Operação	T.M.	C.M.	P.C.
out	$O(1)$	$O(n)$	1
rdp	$O(1)$	$O(n)$	2
inp	$O(n)$	$O(n)$	$6/8$

Tabela 1: Custo das operações do espaço de tuplas.

passos de comunicação extras (ver figura 2). Uma importante característica dos protocolos baseados em quóruns é sua complexidade de mensagens linear ($O(n)$), que em geral permite uma utilização mais racional da rede de comunicação.

5. Trabalhos Relacionados

Em face a diversidade de novas aplicações e modelos de computação distribuída, dois modelos de coordenação que apresentam alto desacoplamento têm se tornado muito populares nos últimos anos: o *Publish/Subscribe* [Eugster et al., 2003] e o generativo [Papadopolous and Arbab, 1998, Cabri et al., 2000].

Os sistemas baseados em *Publish/Subscribe* destacam-se em cenários de larga escala onde podem se valer de redes sobrepostas para a distribuição de mensagens entre os processos. Já o modelo generativo, se mostra atraente pela sua simplicidade de programação (apenas 5 primitivas), seu poder (memória associativa) e sua capacidade de sincronizar processos distribuídos.

Existem várias propostas para replicação de espaço de tuplas. Algumas se baseiam em replicação ativa [Bakken and Schlichting, 1995] e outras em sistemas de quóruns [Xu and Liskov, 1989], entretanto, nenhuma delas considera a ocorrência de faltas bizantinas. O trabalho apresentado em [Chiba et al., 1992] explora a semântica da aplicação executada sobre o espaço de tuplas, definindo diferentes protocolos para manutenção da consistência na replicação considerando os padrões de comunicação esperados durante a execução da aplicação. Esta estratégia não pode ser usada em sistemas onde os processos estão sujeitos a faltas bizantinas, uma vez que não se pode presumir que um processo faltoso seguirá algum padrão de comunicação da aplicação.

Atualmente, uma série de trabalhos têm se concentrado na integração de mecanismos de segurança no modelo generativo. Esta preocupação é legítima na medida em que o modelo vem sendo cada vez mais utilizado em ambientes "hostis" como a Internet. Dentre as propostas presentes na literatura, algumas procuram reforçar políticas de segurança pré-estabelecidas de acordo com o comportamento esperado da aplicação executada [Minsky et al., 2000], outras no entanto se concentram na manutenção da confidencialidade e a integridade das tuplas [De Nicola et al., 1998, Vitek et al., 2003, Busi et al., 2003]. Dentre estes trabalhos, nenhum considera a disponibilidade do espaço de tuplas, objetivo principal dos mecanismos de tolerância a faltas, e muito menos aplicam qualquer abordagem mais robusta relacionada a tolerância a intrusões.

6. Considerações Finais e Trabalhos Futuros

Os requisitos dos modernos sistemas distribuídos abertos, como as redes *ad hoc* e as grades computacionais, têm exigido modelos cada vez mais desacoplados de comunicação. Este trabalho apresenta uma proposta de modelo de coordenação generativa (baseada em espaço de tuplas) que tolera faltas bizantinas a partir da aplicação de replicação por sistemas de

quóruns bizantinos. Até onde sabemos, nossa proposta é a primeira a oferecer este nível de segurança de funcionamento, e nesse fato reside sua maior contribuição. Adicionalmente, os protocolos apresentados definem um objeto de memória compartilhada estritamente mais forte (em termos da hierarquia livre de espera [Herlihy, 1991]) que os registradores usuais implementados sobre sistemas de quóruns.

A construção apresentada neste artigo forma a base para nosso trabalho em coordenação desacoplada tolerante a intrusões para sistemas abertos. Nesse contexto, os trabalhos futuros seguem em várias direções: implementação e simulação dos protocolos (atividade já em andamento), implementação dos protocolos sobre outros tipos de quóruns que provêem um melhor balanceamento de carga como os baseados em grades e partições [Malkhi and Reiter, 1998] e finalmente, a extensão dos protocolos apresentados considerando temas relacionados a segurança (e tolerância a intrusões) como a provisão de confidencialidade das tuplas (utilizando criptografia de limiar).

Referências

Bakken, D. E. and Schlichting, R. D. (1995). Supporting fault-tolerant parallel programing in linda. *IEEE Transactions on Parallel and Distributed Systems*, 6(3):287–302.

Bessani, A. N., da Silva Fraga, J., and Lung, L. C. (2005a). Coordenação desacoplada tolerante a faltas bizantinas. http://www.das.ufsc.br/∼neves/reports/2005-2.pdf.

Bessani, A. N., da Silva Fraga, J., and Lung, L. C. (2005b). O confeiteiro bizantino: Exclusão mútua em sistemas abertos sujeitos a faltas bizantinas. In *Anais do 23o. Simpósio Brasileiro de Redes de Computadores - SBRC 2005*, Fortaleza, CE, Brasil.

Busi, N., Gorrieri, R., Lucchi, R., and Zavattaro, G. (2003). Secspaces: a data-driven coordination model for environments open to untrusted agents. In Brogi, A. and Jacquet, J.-M., editors, *Electronic Notes in Theoretical Computer Science*, volume 68. Elsevier.

Cabri, G., Leonardi, L., and Zambonelli, F. (2000). Mobile agents coordination models for internet applications. *IEEE Computer*, 33(2):82–89.

Carriero, N. and Gelernter, D. (1989). Linda in context. *Communications of the ACM*, 32(4):444–458.

Castro, M., Rodrigues, R., and Liskov, B. (2003). Base: Using abstraction to improve fault tolerance. *ACM Transactions Computer Systems*, 21(3):236–269. A preliminar version appeared *18th Symposium on Operating Systems Principles*, 2001.

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

Chiba, S., Kato, K., and Masuda, T. (1992). Exploiting a weak consistency to implement distributed tuple space. In *Proceedings of the 12th International Conference on Distributed Computing Systems - ICDCS'92.*, pages 416–423, Yokohama, Japan. IEEE Computer Society Press.

De Nicola, R., Ferrari, G. L., and Pugliese, R. (1998). Klaim: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330.

Doudou, A., Garbinato, B., Guerraoui, R., and Schiper, A. (1999). Muteness failure detectors: Specification and implementation. In *Proceedings of the 3rd European Dependable Computing Conference*. Springer-Verlag.

Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131.

Fischer, M. J., Lynch, N. A., and Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382.

Fraga, J. and Powell, D. (1985). A fault- and intrusion-tolerant file system. In *Proceedings of the 3rd International Conference on Computer Security*, pages 203–218.

Gelernter, D. (1985). Generative communication in linda. *ACM Transactions on Programing Languages and Systems*, 7(1):80–112.

Gelernter, D. and Carriero, N. (1992). Coordination languages and their significance. *Communications of ACM*, 35(2):96–107.

Herlihy, M. (1991). Wait-free synchronization. *ACM Transactions on Programing Languages and Systems*, 13(1):124–149. Dijkstra Prize 2003 winner.

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

Lehman, T. J., Cozzi, A., Xiong, Y., Gottschalk, J., Vasudevan, V., Landis, S., Davis, P., Khavar, B., and Bowman, P. (2001). Hitting the distributed computing sweet spot with TSpaces. *Computer Networks*, 35(4):457–472.

Lynch, N. A. (1996). *Distributed Algorithms*. Morgan Kauffman, San Francisco - California.

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

Martin, J.-P., Alvisi, L., and Dahlin, M. (2001). Small byzantine quorum systems. In *Dependable Systems and Networks, DSN 01*. IEEE Computer Society.

Martin, J.-P., Alvisi, L., and Dahlin, M. (2002). Minimal Byzantine storage. In *Distributed Computing, 16th international Conference, DISC 2002*, volume 2508 of *LNCS*, pages 311–325. Springer-Verlag.

Minsky, N. H., Minsky, Y. M., and Ungureanu, V. (2000). Making tuple spaces safe for heterogeneous distributed systems. In *Proceedings of the 2000 ACM symposium on Applied computing*, pages 218–226. ACM Press.

Papadopolous, G. and Arbab, F. (1998). Coordination models and languages. In *The Engineering of Large Systems*, volume 46 of *Advances in Computers*. Academic Press.

Rivest, R. L., Shamir, A., and Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126.

Sun Microsystems (2003). Javaspaces service specification. Disponível em http://www.jini.org/nonav/standards/davis/doc/specs/html/js-spec.html.

Veríssimo, P., Neves, N. F., and Correia, M. P. (2003). Intrusion-tolerant architectures: Concepts and design. Technical Report DI-FCUL TR-03-5, Universidade de Lisboa, Lisboa, Portugal.

Vitek, J., Bryce, C., and Oriol, M. (2003). Coordination processes with secure spaces. *Science of Computer Programming*, (46):163–193.

Xu, A. and Liskov, B. (1989). A design for a fault-tolerant, distributed implementation of linda. In *Proceedings of the 19th Symposium on Fault-Tolerant Computing - FTCS'89.*, pages 199–206. IEEE Computer Society Press.

Avaliando aspectos de tolerância a falhas em protocolos de roteamento para redes de sensores sem fio*

Daniel F. Macedo[1], Luiz H. A. Correia[1,2], Aldri L. dos Santos[1,3], Antonio A. F. Loureiro[1], José Marcos S. Nogueira[1†]

[1]Dep. de Ciência da Computação
Univ. Federal de Minas Gerais
Belo Horizonte-MG, Brasil

[2]Dep. de Ciência da Computação
Univ. Federal de Lavras
Lavras-MG, Brasil

[3]Dep. de Computação
Univ. Federal do Ceará
Fortaleza-CE, Brasil

{damacedo,lcorreia,aldri,loureiro,jmarcos}@dcc.ufmg.br

Abstract. *Fault tolerance is an essential requirement in the design of protocols and applications for Wireless Sensor Networks (WSNs) since communication and hardware failures are frequent. In this paper we studied the resilience of routing protocols for continuous data dissemination WSNs in face of faults. The main causes of silent failure are presented, including some security attacks. Those failures are classified according to extension and persistence, and such classification is used to evaluate routing protocols for continuous data dissemination networks. Results show that failures under a large region of the network are the most damaging. The paper also shows how routing protocols may save energy by temporarily turning off disconnected nodes.*

Resumo. *Tolerância a falhas é um requisito essencial para o projeto de protocolos e aplicações para Redes de sensores sem fio (RSSF), pois falhas de hardware e comunicação são frequentes. Neste trabalho estudamos o comportamento de protocolos de roteamento para redes de disseminação contínua de dados perante a ocorrência de falhas. Apresentamos os principais agentes causadores de falhas silenciosas, incluindo ataques de segurança. Classificamos estas falhas quanto a extensão e persistência, e utilizamos esta classificação para avaliar, via simulação, protocolos de roteamento para redes de disseminação contínua de dados. Verificamos que falhas em grandes regiões da rede são o tipo mais prejudicial, e mostramos como protocolos de roteamento podem economizar energia desligando temporariamente nós isolados da rede.*

1. Introdução

Redes de Sensores Sem Fio (RSSF) são uma subclasse das tradicionais redes ad hoc sem fio. As RSSF são formadas por elementos de rede chamados de nós sensores, que são dispositivos compactos compostos de sensores, processador, rádio, memória e bateria [1]. Esses nós enviam os dados coletados para um Ponto de Acesso (PA), responsável por repassar os dados ao usuário final. Ao contrário das redes ad hoc tradicionais, em geral não é possível trocar ou recarregar a fonte de energia dos nós devido à sua grande quantidade ou às dificuldades impostas pelo ambiente. Assim, o consumo de energia é um fator crítico em RSSF, o que tem motivado o desenvolvimento de protocolos específicos, que tomam decisões procurando otimizar o consumo de energia.

Tolerância a falhas é um dos componentes que constituem um sistema confiável (*dependable*) [2], que é um dos grandes desafios da computação atual [3]. Nas RSSF, falhas são frequentes, e ocorrem em virtude de eventos como a destruição de nós, degradação da qualidade do enlace, entre outros. Visto que essas redes podem ser empregadas em ambientes hostis, como áreas de desastre, os nós podem ser destruídos a qualquer momento, seja por deslizamento, queda de árvores ou prédios, enchentes ou outros agentes naturais. Falhas também ocorrem na comunicação, devido a interferências ocorridas por modificações no clima ou na movimentação de objetos no espaço sensoriado, que bloqueiam o sinal transmitido, bem como agentes maliciosos, que têm como objetivo degradar o serviço da rede. Por se tratar de um ambiente altamente propenso a

*O presente trabalho foi realizado com apoio do CNPq, uma entidade do Governo Brasileiro voltada ao desenvolvimento científico e tecnológico. Processo 55.2111/2002-3.

†Em período sabático nas universidades de Evry e UPMC/Paris6/Lip6, França.

falhas, e considerando o alto grau de interação entre os elementos (os nós em RSSF operam de forma colaborativa), o software dos nós está sujeito a erros devido ao mal funcionamento de outros nós da rede. Assim, protocolos e aplicações em RSSF devem ser desenvolvidos considerando a ocorrência frequente de falhas.

Em protocolos de roteamento as falhas se manifestam como rotas interrompidas e ocorrem devido a erros na comunicação ou defeitos no hardware. Ao detectar um falha na rota, o protocolo de roteamento deve identificar uma rota operacional, permitindo assim que o tráfego entre dois nós seja restaurado. Falhas no roteamento em RSSF são mais graves que em redes ad hoc. Em geral nas RSSF todos os dados são endereçados para um ponto de acesso. Assim, uma rota danificada pode afetar um grande número de fluxos de dados. Em uma rede ad hoc, entretanto, um nó pode enviar dados para qualquer outro nó da rede. Assim, caso um nó falhe, apenas as rotas que dependiam deste nó estarão falhas.

Em RSSF o fluxo de dados segue um padrão, isto é, os dados são processados localmente e enviados para o PA. Este envio pode ser periódico ou esporádico, caracterizando dois tipos de fluxo de dados [4]. Nas *redes dirigidas a eventos*, o envio de dados é ocasional, ocorrendo somente quando verificada uma determinada condição (chamada de evento). Redes dirigidas a evento são utilizadas na localização de animais silvestres, detecção de intrusão, monitoramento de queimadas, entre outros. Nas *redes de disseminação contínua*, os nós enviam mensagens em intervalos regulares para um PA, relatando as leituras atuais dos seus sensores. Em tais redes é possível montar um "mapa" do estado atual da região monitorada, sendo este utilizado para análise de variações espaciais e temporais de eventos. Aplicações dessas redes incluem estudos ambientais, sistemas de tráfego inteligente, monitoração de plantas industriais, entre outros.

Devido às diferenças nas características de tráfego, os protocolos de roteamento são implementados para satisfazer a apenas uma classe de rede. Redes de disseminação contínua de dados tendem a utilizar protocolos pró-ativos, pois a todo momento os nós enviam dados para um PA. Em redes dirigidas a eventos os protocolos são reativos, assim as rotas são construídas somente na ocorrência de um evento de interesse, e apenas na região onde o evento ocorreu. Como os eventos tendem a ser raros e isolados em uma região, a reconstrução periódica de rotas não é recomendável devido ao alto custo de energia associado. O mesmo ocorre com os métodos de tolerância a falhas. Em redes de disseminação contínua, mecanismos pró-ativos se justificam pelo grande volume de dados, enquanto que em redes baseadas em eventos os mecanismos de tolerância a falhas tendem a operar somente quando houver um fluxo de dados.

Neste trabalho estudamos o comportamento de protocolos de roteamento para redes de disseminação contínua de dados, tendo em vista falhas de comunicação e de *hardware* onde o nó não envia dados durante a ocorrência da falha (*falhas silenciosas*). Identificamos os principais agentes causadores de falhas, e caracterizamos as falhas de acordo com a sua extensão e persistência. Esta caracterização é utilizada para extrair características comuns das falhas, e facilitar a avaliação dos protocolos.

O texto está organizado da seguinte forma. A Seção 2 apresenta os trabalhos relacionados. A Seção 3 apresenta uma visão geral dos protocolos avaliados, descrevendo o seu funcionamento e algoritmos de tolerância a falhas. A Seção 4 expõe os agentes causadores de falhas silenciosas em RSSF. A Seção 5 define uma classificação para estas falhas, que é utilizada na avaliação dos protocolos, apresentada na Seção 6. Por fim, a Seção 7 apresenta as conclusões e trabalhos futuros.

2. Trabalhos Relacionados

Avizienis et al. apresentam uma taxonomia de falhas, que engloba questões de segurança [5]. Esta taxonomia considera os desafios e as questões atualmente encontrados nos sistemas atuais. Hollick et al. apresentam os desafios correntes e futuros na tolerância a falhas em RSSF, redes ad hoc e redes celulares, e listam as características necessárias para o desenvolvimento de protocolos confiáveis para estas redes [6]. Koushanfar et al. apresentam uma visão geral de tolerância a falhas em RSSF considerando elementos de hardware, e por fim resumem as técnicas correntes de detecção de falhas bizantinas em sensores, utilizando métodos de correlação [7].

Os primeiros protocolos propostos para RSSF [8, 9] se preocupam com falhas devido ao esgotamento da bateria, e propõem mecanismos para aumentar a vida do nó e distribuir a energia gasta. Estes protocolos não tratam da falha de nós, um evento frequente em RSSF. Outros protocolos se preocupam com falhas na comunicação ocasionadas por quebra de nós, amenizando este problema por uso de múltiplas rotas na transmissão de dados [10, 11]. Visto que cópias dos dados são enviados por múltiplos caminhos, os dados possuem uma maior probabilidade de serem recebidos pelo PA, ao custo de um maior consumo de energia. Um estudo do aumento da probabilidade de recepção de um pacote de acordo com o grau de similaridade das rotas é apresentado por Ganesan et al. [12]. O estudo demonstrou que rotas parcialmente disjuntas são tão eficazes quanto rotas totalmente disjuntas, e possuem um menor custo para serem estabelecidas.

O envio de múltiplas cópias de dados tem um custo elevado, assim é desejável manter apenas uma rota de boa qualidade. De Couto et al. sugerem uma modificação no protocolo de roteamento ad hoc DSR (*Dynamic Source Routing*) para que este considere a qualidade do enlace de uma rota [13], amenizando falhas de comunicação. O protocolo DSR, ao propagar uma requisição de estabelecimento de rota, também propaga um valor acumulado que indica a qualidade do sinal na rota. O nó utiliza sempre a rota com a maior qualidade, evitando enlaces com comunicação intermitente. Alec Woo et al. [14] propuseram um mecanismo para protocolos de roteamento em RSSF que realiza o processamento localmente. Ambas as soluções, entretanto, contemplam um conjunto restrito das falhas em RSSF.

Dado a ocorrência de uma rota falha, é necessário identificar uma rota alternativa. Vieira et al. identificaram duas soluções para a falha de nós devido ao esgotamento de energia [15]. Na primeira, chamada de *Smart-Sink*, o PA notifica aos nós que a rota deve ser modificada, pois um dos nós que compõem a rota possui baixa energia. Na segunda, chamada de *lista de padrastos*, um nó possui uma lista de rotas sobressalentes ao PA. No caso de falha da rota padrão, uma das rotas sobressalentes é utilizada. Ambas as abordagens funcionam apenas para falhas locais, e necessitam de um mecanismo de detecção de falhas, que não é tratado no artigo. Khanna et al. apresentam um mecanismo de rotas alternativas para o protocolo SPIN [16]. Este trabalho apresenta apenas resultados analíticos, não existindo uma definição clara do modelo de falhas tratado. Neste artigo, por outro lado, definimos o modelo de falhas considerado e realizamos uma análise extensiva por simulação dos protocolos avaliados.

Uma outra maneira de tornar um sistema robusto é a prevenção de falhas. Esta abordagem é pouco considerada em RSSF, pois os dispositivos são de baixo custo e possuem capacidades restritas, dificultando o uso de mecanismos para diagnóstico de um nó. Mecanismos indiretos de diagnóstico do nó, como a análise dos valores encontrados nos sensores, podem indicar futuras falhas [17].

3. Os Protocolos Avaliados

Avaliamos o comportamento de três protocolos de roteamento existentes na literatura, TinyOS Beaconing, EAD e PROC, considerando a ocorrência de falhas [11, 18, 19]. Escolhemos esses protocolos por serem desenvolvidos para redes de disseminação contínua de dados e apresentarem diferentes níveis de tolerância a falhas. O TinyOS Beaconing é um protocolo simples, sendo o protocolo padrão da plataforma Mica Motes [20]. O protocolo EAD é um protocolo de roteamento que tem como objetivo a economia de energia. O EAD permite observar como políticas de economia de energia se comportam frente a aspectos de tolerância a falhas. O terceiro protocolo, PROC, é um protocolo que possui mecanismos internos de tolerância a falhas, mostrando em seus resultados os benefícios de prover mecanismos de tolerância a falhas mais elaborados.

O protocolo TinyOS Beaconing recria periodicamente uma árvore de roteamento de menor caminho, com raiz no PA. Para criar essa árvore, o PA envia periodicamente uma mensagem *beacon* para a rede, determinando a distância em saltos dos nós até ele. A seleção das rotas também leva em conta a confiabilidade do enlace, calculada pelo número de pacotes corretamente enviados pelos nós vizinhos. Para reduzir o número de retransmissões, apenas nós com enlaces confiáveis são usados no roteamento. O TinyOS Beaconing utiliza o mecanismo de recriação

periódica das rotas como estratégia de tolerância a falhas.

O protocolo de roteamento EAD (*Energy-Aware Distributed routing*), proposto por Boukerche et al., cria uma árvore de roteamento com o objetivo de maximizar o número de nós folha [18]. Os nós folha não roteiam dados, e portanto podem manter o seu rádio desligado por períodos prolongados de tempo, economizando energia. O EAD atrasa a transmissão de dados em um intervalo proporcional à energia residual, que diminui significativamente a quantidade de colisões. Como ocorre no TinyOS Beaconing, o protocolo depende da reconstrução periódica de rotas para identificar rotas falhas. No EAD, entretanto, o fluxo de dados é concentrado em um número reduzido de nós, logo a ocorrência de falhas nestes nós é mais grave que falhas em nós folha.

O protocolo PROC (*Proactive ROuting with Coordination*) tem como meta diminuir o consumo de energia e maximizar o tempo de vida da rede [19]. Como o TinyOS Beaconing e o EAD, o PROC cria uma árvore de roteamento, chamada de *backbone*. A construção do *backbone* é regida pela aplicação, que define quais nós são os mais apropriados para rotear dados. O *backbone* é reconstruído periodicamente por um processo iniciado pelo PA. O protocolo possui um mecanismo de reconstrução local de rotas, que utiliza as mensagens de confirmação de recepção da camada de controle de acesso ao meio (MAC) para identificar nós falhos. Ao detectar que o próximo salto na rota não está confirmando a recepção das mensagens, o nó recalcula a sua rota. A falha de nós do *backbone* pode comprometer a aquisição de dados de uma região. Para solucionar este problema, o PROC monitora o próximo salto da sua rota para reconstruí-las dinamicamente.

3.1. Mecanismos de Tolerância a Falhas

Para melhor caracterizar o comportamento dos protocolos avaliados, é importante conhecer como cada protocolo lida com a ocorrência de falhas. Esta seção descreve os mecanismos de tolerância a falhas empregados nos protocolos avaliados.

A recriação periódica de rotas é um mecanismo de tolerância a falhas empregado pelos três protocolos. A cada nova execução do algoritmo de recriação de rotas dos protocolos EAD, PROC e TinyOS Beaconing, as rotas são completamente reconstruídas, utilizando somente os nós ativos. Este processo ocorre da seguinte forma: O PA envia uma mensagem indicando que as rotas devem ser recriadas. Cada nó repassa esta mensagem para os seus vizinhos em *broadcast*, permitindo que os nós identifiquem quais vizinhos estão ativos. Ao utilizarem como rotas os nós vizinhos que repassaram a última mensagem de recriação de rotas enviada pelo PA, os nós evitam o uso de vizinhos falhos.

O intervalo entre cada recriação de rotas deve ser ajustado de acordo com o grau de tolerância a falhas e o consumo de energia desejados. Por ser um processo que requer o envio de mensagens para toda a rede, a recriação de rotas envia um grande número de mensagens. Do ponto de vista de consumo de energia, este processo deve ocorrer com a menor frequência possível, mas do ponto de vista de falhas, quanto mais frequente for a atualização de rotas, mais rápido as falhas serão contornadas.

O PROC permite que o roteamento recupere rotas falhas mais rapidamente do que os outros protocolos, pois utiliza um mecanismo de monitoração da atividade dos sensores equivalente a mensagens de *ping-pong*. Para cada pacote de dados enviado pelo PROC (ping), o receptor deve retornar um quadro de confirmação (pong). Não são enviadas mensagens adicionais, pois o PROC utiliza os quadros de confirmação de recebimento (ACK) do protocolo MAC. Um contador registra quantos ACKs consecutivos foram perdidos. Quando o contador ultrapassa um certo limiar, o nó assume que seu pai está falho, e recalcula as suas rotas. Este mecanismo permite que o nó identifique rotas falhas antes da recriação periódica de rotas, aumentando a resiliência da rede. O uso de ACKs do protocolo de enlace dispensa o envio de uma mensagem adicional, economizando energia. Devido ao pequeno tamanho dos quadros de dados e às restrições de energia, em geral protocolos MAC para RSSF evitam o uso de quadros de controle. Entretanto, estudos de Polastre et al. mostraram que o aumento da latência e do consumo de energia são mínimos para o protocolo B-MAC quando os quadros de confirmação são ativados [21], justificando assim o seu uso para prover tolerância a falhas.

A falta de confirmação ocorre por dois motivos: A mensagem original não foi recebida pelo emissor, ou esta foi recebida mas sofreu erros na transmissão. Assim, é importante determinar

se um ACK não foi recebido porque o destinatário da mensagem está falho ou se houve um erro de transmissão. Utilizamos a probabilidade de que um quadro seja perdido (PER – *Packet Error Rate*)[1] para determinar o número mínimo de quadros consecutivos não recebidos que identificarão a falha de um nó. O número de quadros consecutivos perdidos que definem uma falha de nós (limiar) deve ser tal que a taxa de falsos positivos (erros de retransmissão considerados como falhas de nós) seja próxima de zero:

$$PER(t)^{limiar} = P, P \approx 0 \quad (1)$$

A escolha de um limiar alto faz com que uma grande quantidade de dados precise ser perdida até que a falha seja detectada, mas a certeza da falha aumenta, enquanto um limiar baixo aumenta a quantidade de falsos positivos, mas diminui o tempo necessário para a detecção de falhas de nós. A probabilidade de que ocorram n quadros consecutivos com erro diminui rapidamente quando aumentamos o número de quadros (n). Para n pequeno, conseguimos facilmente distinguir entre erros de transmissão e falhas de nós. Para efeito de comparação, o rádio CC1000, utilizado pelos nós Mica2 [20], possui taxa de erro por bit típica de 10^{-3} [22]. Na seção 6, onde avaliamos o comportamento de protocolos, utilizamos para o PROC um limiar igual a dois quadros, obtendo uma probabilidade de 0.001% de falsos positivos.

4. Falhas em RSSF

Esta seção identifica as principais causas de falhas de comunicação em RSSF. Neste estudo não consideramos falhas de comunicação em decorrência de resultados errados (falhas erráticas) produzidas pelos nós sensores. A corretude das mensagens de roteamento é assegurada por códigos de detecção de erro e mecanismos de verificação formal da especificação, garantindo que o roteamento não apresente falhas erráticas. Assumindo a corretude do protocolo e suas mensagens, podemos nos concentrar apenas em falhas silenciosas, ocasionadas pela não recepção de pacotes ou erros de hardware nos nós sensores. As seguintes falhas foram identificadas:

Fenômenos atmosféricos – Mudanças nas condições atmosféricas alteram a propagação do sinal, causando um aumento na taxa de erros da comunicação, proporcionado pela atenuação do sinal transmitido. Condições do ambiente como umidade, temperatura, entre outros, modificam a qualidade dos enlaces. Como as características do ambiente são dinâmicas, a qualidade da comunicação varia com o tempo.

Fontes móveis de interferência – Aparelhos que operam em faixas de frequência próximas às utilizadas em RSSF, ou mesmo veículos, animais e pessoas, podem gerar interferência na comunicação. Por operarem em frequências na faixa ISM (*Industrial, Scientifical and Medical*), que não requer licença para operação, RSSF estão expostas a interferência de outros aparelhos que operam nesta faixa de frequência. Para baratear o custo dos nós sensores, o rádio geralmente emprega um único canal, e possui modulação fixa. Estas limitações impedem o uso de mecanismos como seleção de canais com menor interferência, saltos de frequência ou troca dinâmica do mecanismo de modulação [23].

Desastres naturais – Nós sensores podem ser depositados ao ar livre ou em regiões de desastre, estando assim expostos a deslizamentos, terremotos e enchentes. Desastres naturais podem ocasionar a destruição dos nós ou a inutilização de componentes do hardware dos nós sensores. Ao contrário das falhas decorrentes das condições atmosféricas, nós falhos devido a desastres naturais permanecem inoperantes.

Quebra acidental – Nós sensores podem ser destruídos acidentalmente, por exemplo devido ao pisoteamento por animais ou à queda de árvores sobre nós sensores. Em geral, os nós sensores estarão espalhados a alguns metros de distância uns dos outros, assim apenas um nó falha por vez.

Bloqueio do processador – Nos sistemas embutidos são utilizados escalonadores de multitarefa cooperativa, ou sistemas de *run to completion*, assim um software defeituoso pode bloquear o processador por tempo indeterminado. Para evitar tais situações, são utilizados temporizadores

[1] $PER(t) = 1 - (1 - BER)^t$, onde t é o tamanho do quadro enviado e BER é a taxa de erro por bit.

chamados de *watchdogs*[2]. Desta forma, o nó estará bloqueado por um tempo finito, e em seguida retornará à operação normal.

Falhas maliciosas – RSSF estão expostas a falhas maliciosas, como decorrentes de ataques de segurança. Nestas falhas, um nó malicioso ou uma entidade externa provocam erros na rede. Este trabalho não aborda técnicas de prevenção a ataques. Entretanto, é possível utilizar mecanismos simples de tolerância a falhas para identificar regiões ou nós sobre ataque e evitá-las [24]. Neste trabalho abordamos apenas mecanismos para contornar alguns ataques de negação de serviço (*ataques de interferência*, de *colisão* e de *sinkhole*). Estes ataques se comportam como falhas silenciosas, que são o escopo do nosso trabalho.

Esgotamento da bateria – O esgotamento da bateria dos nós sensores pode gerar uma falha de comunicação. O emprego de nós em áreas de difícil acesso ou o número de nós empregados pode tornar inviável o recarregamento de baterias. Por possuírem hardware limitado, os nós sensores atuais não permitem a aferição confiável do nível corrente de energia, desta forma o nó sensor não tem como notificar aos nós vizinhos a iminência da sua falha.

Falhas por esgotamento da bateria não são consideradas devido à dificuldade de se modelar tais falhas. O esgotamento da energia em geral ocorrerá simultaneamente para uma grande quantidade de nós, uma vez que os protocolos tendem a balancear o consumo entre os nós para maximizar o tempo de vida da rede [8]. Nós ativos assumirão as tarefas dos nós indisponíveis, aumentando a sua carga e consequentemente o consumo de energia. Com isso, a rede irá rapidamente deteriorar, em um efeito em cascata. Para se manter conectada, a rede utilizará mecanismos de auto-configuração para ajustar o alcance do rádio dos nós restantes para que estes possam alcançar outros nós ainda ativos.

5. Agrupamento das Falhas

Esta seção apresenta um agrupamento das falhas descritas na seção 4 de acordo com as suas características. Esse agrupamento tem como objetivo facilitar o estudo de falhas em RSSF, sendo resumido na Tabela 1. As falhas são caracterizadas quanto a sua persistência e extensão:

Persistência – Indica se o nó retornará a operar corretamente após um período de tempo falho (*falhas transientes*), ou se a falha é *permanente* [5]. Do ponto de vista do roteamento, falhas transientes são aquelas em que a falha pode ser contabilizada em minutos, enquanto falhas permanentes podem ser contabilizadas em horas. Consideramos que situações de falha devido às mudanças no clima, por exemplo, serão caracterizadas como permanentes.

Extensão – Indica o número de nós afetados. As falhas podem ser *isoladas*, no caso da falha de um único nó, ou *agrupadas*, onde um conjunto de nós falha. O último aumenta a gravidade da falha, uma vez que grande parte da vizinhança de um nó se tornará falha, diminuindo o número de vizinhos que poderão rotear dados. A Figura 1 exemplifica esta classificação (setas representam as rotas dos nós).

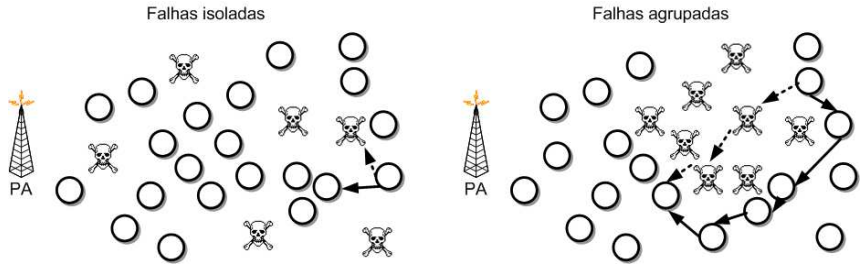


Figura 1: Exemplo de falhas de nós, classificadas quanto a extensão.

As falhas maliciosas decorrentes de ataques de colisão e de *sinkhole* variam de duração de acordo com a intenção do atacante, podendo ser breves para evitar detecção, ou prolongadas, aumentando o estrago causado. Assim, classificamos essas falhas tanto como transientes quanto

[2]O *watchdog* é um temporizador a ser reiniciado periodicamente pela aplicação, ou o processador é reiniciado.

como permanentes. Ataques de interferência, por outro lado, serão permanentes, pois estes são mais efetivos se empregados por um período prolongado, mesmo ocorrendo a detecção do ataque.

Causa da falha	**Persistência**	**Extensão**
Fenômenos atmosféricos	permanente	agrupadas
Fontes móveis de interferência	transiente	isolada
Desastres naturais	permanente	agrupadas
Quebra acidental	permanente	isolada
Bloqueio do processador	transiente	isolada
Ataques de interferência	permanente	agrupadas
Ataques de colisão	ambos	isolada
Ataques de *sinkhole*	ambos	isolada

Tabela 1: Caracterização das falhas de acordo com a sua causa.

6. Avaliação por Simulação

Avaliamos o desempenho dos três protocolos utilizando o simulador NS-2 [25]. Simulamos uma rede homogênea, composta por nós sensores com configuração próxima aos nós sensores da família Mica 2, rodando o sistema operacional TinyOS [20]. Escolhemos esta plataforma devido à sua grande aceitação na comunidade científica e ao elevado número de redes em operação que a utilizam. Simulamos uma aplicação com características de tráfego similares à rede empregada na ilha de Great Duck para estudos do ecosistema e comportamento de aves [17]. Nesta rede, cada nó sensor envia mensagens de dados de 36 bytes a cada 70s. Estas mensagens são enviadas para o PA, que disponibiliza os dados para análise.

O protocolo de acesso ao meio empregado é uma versão modificada da implementação do IEEE 802.11, que emula o comportamento do protocolo B-MAC [21]. Limitamos a banda em 12kbps, como ocorre no TinyOS, e ajustamos os parâmetros do rádio de acordo com as especificações do rádio CC1000 [22], utilizado nos nós Mica2. Definimos o intervalo de recriação dos protocolos EAD e TREE (uma versão do TinyOS Beaconing sem o cálculo de confiabilidade do canal) em 120s, enquanto no protocolo PROC utilizamos 180s. Estes valores, determinados empiricamente, foram utilizados por apresentarem os melhores valores para cada protocolo avaliado [19].

A rede considerada é formada por 150 nós inseridos aleatoriamente em uma área quadrada, com 70m de lado. O PA se encontra no canto da rede em todos os cenários de simulação, maximizando o número de saltos. A rede funciona sem falhas durante 1500s, para que os nós cheguem ao seu estado estacionário (verificado empiricamente). Neste momento ocorre uma falha, e a simulação continua por mais 1500s, permitindo que o protocolo de roteamento se recupere da falha. Nos cenários em que ocorrem falhas isoladas, os nós que irão falhar são escolhidos aleatoriamente. Para falhas agrupadas, é definido um ponto central, e todos os nós que estão a uma distância máxima r_{max} do ponto irão falhar.

As métricas avaliadas são: *taxa de entrega média fim a fim* (porcentagem do total de pacotes recebidos corretamente pelo PA); *latência média*; *distância média em saltos até o PA*; *vazão*; e *consumo médio de energia*. O consumo médio é calculado somente para os nós que estão ativos ao final da simulação. A análise dos resultados enfatiza a energia média consumida, taxa de entrega média e vazão média, pois estas são as métricas mais importantes quando consideramos tolerância a falhas em RSSF. Todos os resultados são obtidos pela média de 33 simulações, e apresentados com intervalo de confiança de 95%.

6.1. Falhas Transientes e Isoladas

As falhas transientes foram avaliadas sobre três dimensões: intervalo de recriação de rotas, tempo da falha e número de nós falhos. A frequência da atualização de rotas influirá no grau de tolerância a falhas do protocolo, pois define o tempo de reação a falhas em protocolos como o EAD e TREE, que dependem da reconstrução de rotas para normalizar o funcionamento da rede. Neste cenário 20 nós falharam durante 120s, e o tempo de ciclo variou de 120 a 300s. Como era

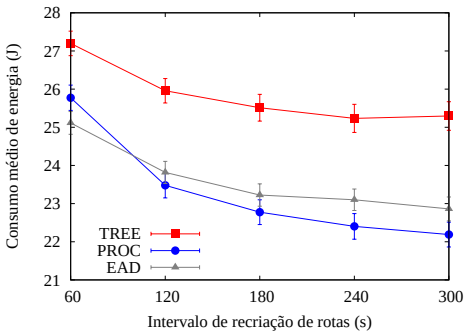


Figura 2: Energia consumida variando o intervalo de recriação de rotas.

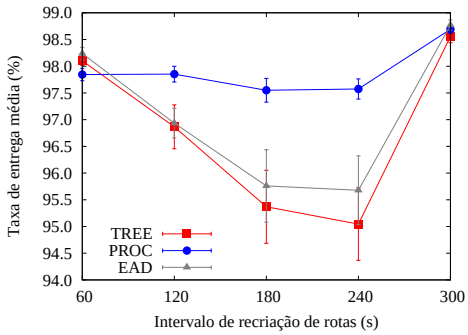


Figura 3: Taxa de entrega média variando o intervalo de recriação de rotas.

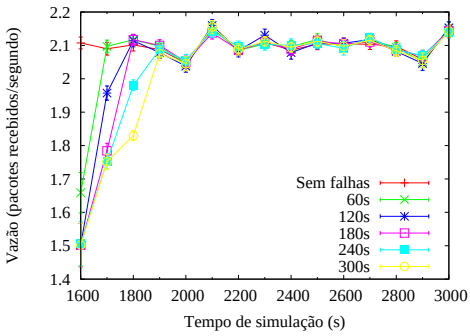


Figura 4: Vazão do PROC variando o tempo de falha.

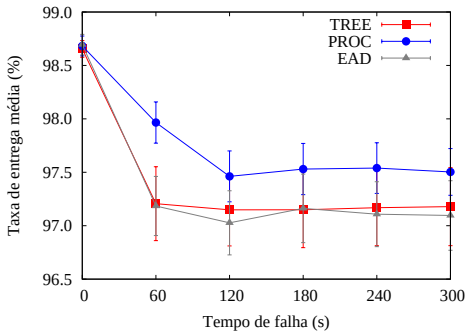


Figura 5: Taxa de entrega variando o tempo de falha.

esperado, os protocolos tendem a consumir mais energia com atualizações de rotas mais frequentes (Figura 2). A taxa de entrega, mostrada na Figura 3, diminui ao aumentarmos o intervalo de recriação de rotas. No PROC a redução é menor, devido ao uso de ACKs para identificar falhas, pois as falhas são corrigidas mais rapidamente. Verificamos um aumento na taxa de entrega para intervalos de 300s, que se deve a uma diminuição na carga da rede, diminuindo o número de pacotes perdidos devido a falta de espaço na fila. Quanto à latência média, todos os protocolos mostraram uma diminuição nesta métrica com o aumento do tempo de recriação de rotas, que ocorre principalmente devido à menor carga imposta à rede.

Em seguida avaliamos como o tempo de falha afeta o desempenho dos protocolos. Para todos os protocolos, verificamos que existe uma queda na vazão durante o tempo de falha, que é recuperada quase completamente após um intervalo de recriação de rotas (Figura 4). Isto ocorre porque os protocolos utilizam como mecanismo de tolerância a falhas a recriação completa das rotas em intervalos regulares de tempo. Novamente, o PROC apresentou uma taxa de entrega média levemente superior (em torno de 0.5%), devido ao uso de ACKs para detectar falhas, como mostrado na Figura 5. O intervalo de recriação de rotas mostrou-se adequado, uma vez que os protocolos EAD e TREE, mesmo sem possuírem mecanismos ativos de detecção de falha como o PROC, obtiveram bons resultados. Como anteriormente, a quantidade de energia consumida por nó diminuiu, uma vez que estes tiveram que rotear menos dados. Dentre os protocolos avaliados, o PROC consumiu menos energia, consumindo em torno de 22 a 23J, enquanto o EAD e TREE consumiram em torno de 4% a 14% mais energia que o PROC, respectivamente.

Finalmente, variamos o número de nós falhos. Simulamos falhas de 25 a 100 nós, com incrementos de 25 nós. Identificamos que todos os protocolos possuem um comportamento similar quando variamos o número de nós falhos. Os protocolos apresentam um declínio de vazão, que é restaurada em até 200s após a falha. A queda na vazão é determinada pelo número de nós falhos, exemplificada pelo comportamento do PROC, mostrado na Figura 6. O mecanismo de identificação de falhas empregado pelo PROC permitiu que este protocolo recuperasse mais rapidamente da falha, desta forma aumentando a sua taxa de entrega em relação aos outros protocolos em torno de 0.5%. Como a falha tem um intervalo de tempo pequeno em relação ao tempo total de simulação, o ganho por uma recuperação mais rápida da falha é amenizado no resultado final.

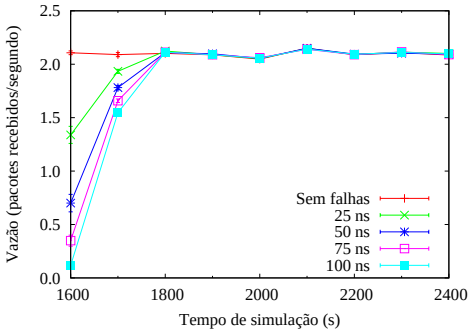


Figura 6: Vazão do PROC variando o número de nós falhos.

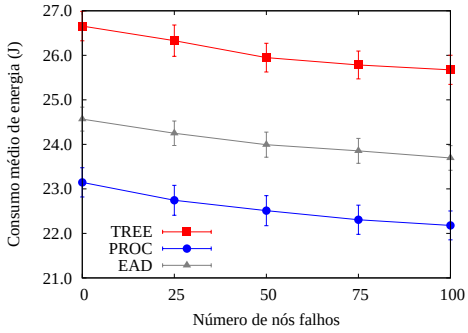


Figura 7: Consumo de energia variando o número de nós falhos.

A latência e o número de saltos médios não sofrem alterações com a falha de nós, entretanto a energia consumida decresce. Isto ocorre devido à diminuição do número de pacotes enviados ao PA, correspondente ao tráfego originado dos nós falhos. A Figura 7 mostra o gráfico do consumo médio de energia dos protocolos. A pequena variação do consumo médio e da taxa de entrega média mostram que o número de nós falhos não influi de forma significativa na rede. Como as falhas são distribuídas pela rede, novas rotas são facilmente encontradas.

6.2. Falhas Permanentes e Isoladas

Neste cenário avaliamos o comportamento dos protocolos na ocorrência de falhas permanentes e isoladas. Realizamos simulações variando o número de nós falhos em 20, 40 e 60 nós. Verificamos que os protocolos se recuperam rapidamente, entretanto a vazão cai após a ocorrência da falha, pois os nós falhos deixam de enviar dados. A desativação dos nós fez com que o número de saltos até o PA aumentasse levemente, em torno de 0.1 saltos para cada 20 nós falhos. A latência média se manteve quase inalterada, mostrando que a diminuição do tráfego de dados compensou o aumento do número de saltos médios. Quanto à taxa de entrega média, verificamos uma queda com o aumento do número de nós falhos, mostrado na Figura 8.

Em comparação com as falhas transientes isoladas, verificamos que as falhas permanentes são mais prejudiciais à rede do que as falhas transientes, pois acarretam uma queda mais significativa no consumo médio de energia e na taxa de entrega média.

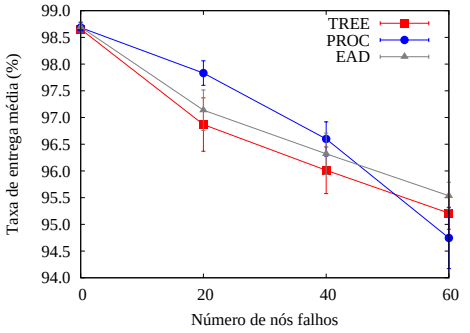


Figura 8: Taxa de entrega média para falhas permanentes isoladas.

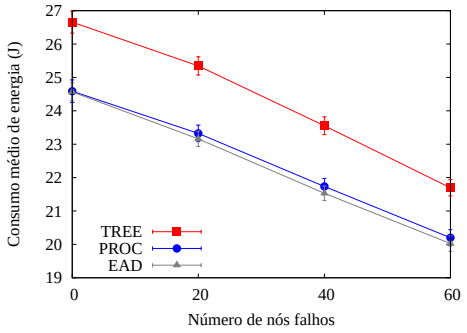


Figura 9: Consumo de energia para falhas permanentes isoladas.

6.3. Falhas Permanentes e Agrupadas

Neste cenário avaliamos o efeito de falhas permanentes agrupadas. O número de nós falhos depende do raio de falha, que variou de 5 a 40m. Os resultados obtidos mostram que este tipo de falha é o mais severo dentre os avaliados, assim realizamos uma análise mais completa.

A taxa de entrega média cai até 9% com o aumento do raio de falha, como mostra a Figura 10. Como nos cenários anteriores, os protocolos se recuperam da falha após a recriação completa das rotas. Neste cenário, entretanto, verificamos uma grande variação da taxa de entrega média, causada por cenários onde ocorrem partições na rede. É sabido que nós próximos ao PA repassam mais dados que os nós distantes, assim falhas nestes nós irão degradar significativamente o serviço

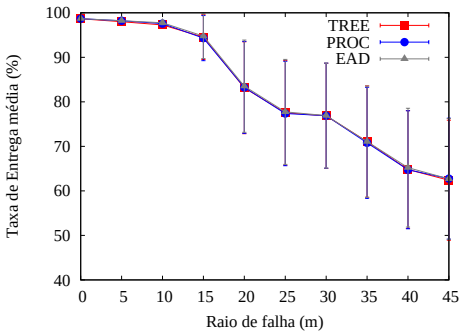


Figura 10: Taxa de entrega para falhas permanentes agrupadas.

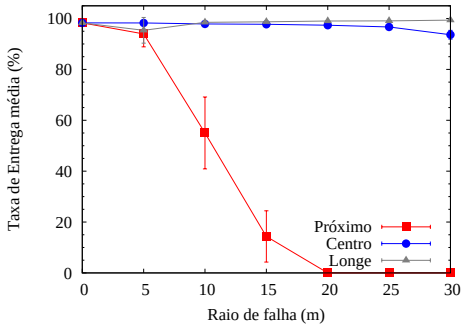


Figura 11: Taxa de entrega para falhas em pontos distintos da rede.

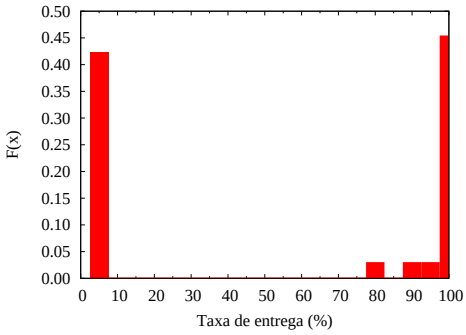


Figura 12: Histograma da taxa de entrega em falhas perto do PA.

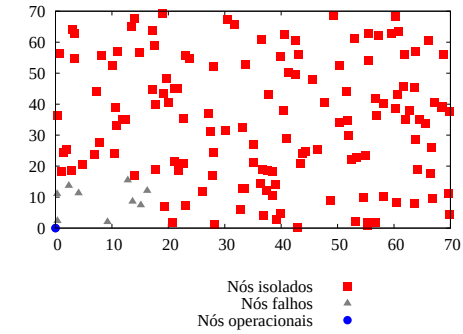


Figura 13: Exemplo de rede particionada.

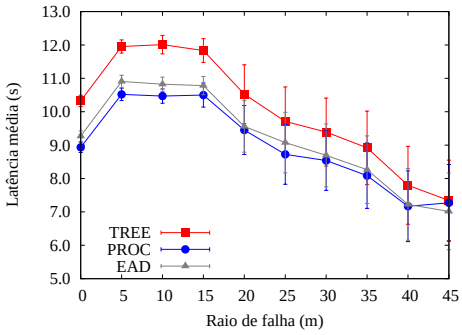


Figura 14: Latência média para falhas permanentes agrupadas.

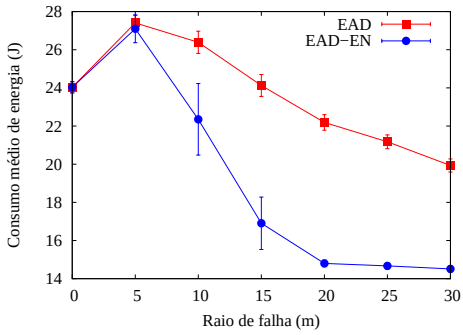


Figura 15: Consumo de energia com mecanismos de desligamento do rádio.

da rede. A Figura 11 exemplifica a taxa de entrega para falhas em pontos distintos da rede. A curva “Próximo” apresenta a taxa de entrega média para falhas que possuem o seu centro em até 17m do PA. A curva “Centro” mostra falhas na região central da rede, enquanto a curva “Longe” apresenta falhas em até 17m do canto oposto ao PA. Os resultados apontam que falhas de nós em pontos não muito próximos do PA pouco afetam a taxa de entrega. Entretanto, falhas próximas ao PA podem degradar substancialmente o desempenho da rede.

Encontramos um intervalo de confiança de até 10% para a taxa de entrega na curva “Próximo” da Figura 11, que verificamos ser ocasionado por partições na rede. Uma análise do histograma da taxa de entrega para a curva “Próximo” (Figura 12), em grupos de 5%, mostra que os resultados estão aglomerados próximo de 5% e de 95%, o que explica o alto intervalo de confiança, e identifica o papel crucial da partição da rede no desempenho dos protocolos neste cenário. As partições ocorrem em cenários como o mostrado na Figura 13, onde a ocorrência de falhas isolou todos os nós operacionais da rede, impedindo a comunicação com o PA. A figura classifica os nós em “Nós falhos” (nós que sofreram falhas), “Nós operacionais” (nós que estabelecem rotas para o PA) e “Nós isolados” (nós que não conseguem estabelecer rotas para o PA). Uma forma para recuperar partições na rede, que necessita de suporte no MAC, é o aumento da potência de transmissão. Com a negociação de uma nova potência de transmissão entre os nós

próximos à região de falhas, a comunicação poderia ser reestabelecida. O roteamento também pode contribuir para amenizar a severidade de uma partição da rede detectando a ocorrência de falhas e adotando medidas de economia de energia.

Verificamos que a latência média (Figura 14) e a distância média em saltos até o PA aumentam quando o raio da falha é pequeno, pois é necessário que as rotas evitem a região falha, para tanto aumentando o caminho percorrido. Para falhas mais extensas, devido a partições na rede, verificamos uma diminuição em ambas as métricas. Como apenas os nós próximos do PA conseguem enviar dados, o número de saltos médios e a latência média diminuem.

A Figura 15 mostra mecanismos de economia de energia que podem ser utilizados na ocorrência de uma partição na rede para diminuir o consumo dos nós. Comparamos o EAD ao EAD-EN, uma versão modificada do EAD, que recusa o envio de mensagens da aplicação caso o nó não tenha recebido mensagens de recriação de rotas em um intervalo de tempo equivalente a dois períodos entre a recriação de rotas (curva "EAD-EN" na figura). Como visto na figura, o uso de técnicas de economia de energia permite uma diminuição substancial no consumo (de 16% a 33% apenas evitando o envio de mensagens), não interferindo em nenhuma outra métrica da rede. Verificamos resultados semelhantes para os outros protocolos avaliados, justificando a implementação destas técnicas em protocolos de roteamento.

7. Conclusões e Trabalhos Futuros

Redes de sensores sem fio são aplicadas em ambientes inóspitos, estando sujeitas a intempéries e variações climáticas, que acarretam em um aumento significativo na frequência e severidade das falhas. Os nós sensores devem se adaptar às condições do ambiente para continuar a prover um serviço dentro dos requisitos de qualidade esperados. Para tanto, é necessário que a rede mantenha rotas dos nós sensores ao ponto de acesso mesmo na presença de falhas e ataques de segurança. Neste artigo caracterizamos os principais agentes causadores de falhas silenciosas nestas redes, e a partir desta caracterização avaliamos o comportamento de protocolos de roteamento na ocorrência de falhas.

Verificamos que os protocolos avaliados apresentam mecanismos de auto-estabilização, como a recriação periódica de todas as rotas, que permitem recuperação em situações de falha. Falhas transientes isoladas e permanentes isoladas são facilmente tratadas com mecanismos de recriação de rotas, e apresentam pouco impacto no funcionamento da rede. Falhas permanentes e agrupadas, entretanto, podem causar grandes perdas de dados, pois podem gerar partições na rede. Mecanismos de tolerância a falhas que tratam falhas permanentes agrupadas devem ser mais agressivos perto do PA, pois falhas nesta região podem comprometer o funcionamento de toda a rede. Uma maneira de abrandar o custo destas falhas é o desligamento temporário de nós que não se comunicam com o PA, o que permite economia significativa de energia em situações onde ocorre uma falha prolongada.

Como trabalhos futuros, iremos identificar como os parâmetros de QoS se comportam na presença de falhas, verificando se os protocolos atuais podem ser considerados confiáveis quando aplicados em redes com altas taxas de falhas.

Referências

[1] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A Survey on Sensor Networks. *IEEE Communications*, 40(8):102–114, 2002.

[2] Benjamin Lussier, Raja Chatila, Felix Ingrand, Marc-Olivier Killijian, and David Powell. On fault tolerance and robustness in autonomous systems. In *3rd IARP-IEEE/RAS-EURON Joint Workshop on Technical Challenges for Dependable Robots in Human Environments*, 2004.

[3] The British Computer Society. Grand Challenges in Computing. http://www.nesc.ac.uk/esi/events/Grand_Challenges/, 2004.

[4] Linnyer Beatrys Ruiz, Antonio A. F. Loureiro, and Jose Marcos Nogueira. Functional and information models for the MANNA architecture. In *GRES03 - Colloque Francophone sur la Gestion de Reseaux et de Services*, pages 455–470, February 2003.

[5] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, 2004.

[6] Matthias Hollick, Ivan Martinovic, Tronje Krop, and Ivica Rimac. A Survey on Dependable Routing in Sensor Networks, Ad hoc Networks, and Cellular Networks. In *Proceedings of the 30th IEEE EUROMICRO Conference 2004*, pages 495–502, Rennes, France, September 2004.

[7] Farinaz Koushanfar, Miodrag Potkonjak, and Alberto Sangiovanni-Vincentelli. Fault tolerance in wireless sensor networks. In *Handbook of Sensor Networks: Compact Wireless and Wired Sensing Systems*. CRC Press, 2004.

[8] Wendi Rabiner Heinzelman and Anantha Chandrakasan and Hari Balakrishnan. Energy-Efficient Communication Protocol for Wireless Microsensor Networks. In *Proceedings of the 33rd Hawaii International Conference on System Sciences*, 2000.

[9] K. Sohrabi and J. Gao and V. Ailawadhi and G. Pottie. Protocols for Self-Organization of a Wireless Sensor Network. *IEEE Personal Communications*, 7(5):16–27, 2000.

[10] Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin, John Heidemann, and Fabio Silva. Directed diffusion for wireless sensor networking. *ACM/IEEE Transactions on Networking*, 11(1):2–16, February 2002.

[11] Chris Karlof, Yaping Li, and Joseph Polastre. ARRIVE: Algorithm for robust routing in volatile environments. Technical Report UCB//CSD-03-1233, University of California, Berkeley, CA, March 2003.

[12] Deepak Ganesan and Ramesh Govindan and Scott Shenker and Deborah Estrin. Highly-Resilient, Energy-Efficient Multipath Routing in Wireless Sensor Networks. *SIGMOBILE Mob. Comput. Commun. Rev.*, 5(4):11–25, 2001.

[13] Douglas S. J. De Couto, Daniel Aguayo, John Bicket, and Robert Morris. A high-throughput path metric for multi-hop wireless routing. In *Proceedings of the 9th ACM International Conference on Mobile Computing and Networking (MobiCom '03)*, San Diego, California, September 2003.

[14] Alec Woo, Terence Tong, and David Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proceedings of the first international conference on Embedded networked sensor systems*, pages 14–27. ACM Press, 2003.

[15] Marcos Augusto M. Vieira, Luis Filipe M. Vieira, Linnyer Beatrys Ruiz, Antonio Alfredo F. Loureiro, Antônio O. Fernandes, José Marcos S. Nogueira, and Diógenes Cecílio da Silva Jr. Como Obter o Mapa de Energia em Redes de Sensores Sem Fio? Uma Abordagem Tolerante a Falhas. In *Anais do 5o. Workshop de Comunicação sem Fio (WCSF)*, pages 183–189, 2003.

[16] Gunjan Khanna, Saurabh Bagchi, and Yu-Sung Wu. Fault tolerant energy aware data dissemination protocol in sensor networks. In *IEEE Dependable Systems and Networks Conference*, June 2004.

[17] R. Szewczyk, J. Polastre, A. Mainwaring, and D. Culler. Lessons from a sensor network expedition. In *Proceedings of the First European Workshop on Sensor Networks (EWSN)*, pages 307–322, Jan 2004.

[18] Azzedine Boukerche, Xiuzhen Cheng, and Joseph Linus. Energy-aware data-centric routing in microsensor networks. In *Proceedings of the 6th international workshop on Modeling analysis and simulation of wireless and mobile systems*, pages 42–49. ACM Press, 2003.

[19] Daniel F. Macedo, Luiz H. A. Correia, Aldri L. dos Santos, Antonio A. Loureiro, and José M. Nogueira. A pro-active routing protocol for continuous data dissemination wireless sensor networks. In *10th IEEE Symposium on Computer and Communications (ISCC)*, Jun 2005.

[20] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and David Culler. TinyOS: An operating system for wireless sensor networks. In W. Weber, J. Rabaey, and E. Aarts, editors, *Ambient Intelligence*. Springer-Verlag, New York, NY, 2004.

[21] Joseph Polastre, Jason Hill, and David Culler. Versatile low power media access for wireless sensor networks. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 95–107. ACM Press, 2004.

[22] CC1000. Chipcom corporation. CC1000 low power FSK transceiver. http://www.chipcom.com, 2004.

[23] Bernhard Walke, Norbert Esseling, Jörg Habetha, Andreas Hettich, Arndt Kadelka, Stefan Mangold, Jörg Peetz, and Ulrich Vornefeld. IP over Wireless Mobile ATM - Guaranteed Wireless QoS by HiperLAN/2. *Proceedings of the IEEE*, 89:21–40, Jan 2001.

[24] Anthony D. Wood and John A. Stankovic. Denial of service in sensor networks. *Computer*, 35(10):54–62, 2002.

[25] NS-2 simulator. http://www.isi.edu/nsnam/ns/, January, 2004.

Roteamento Tolerante a Falhas Baseado em Caminhos Robustos

Jonatan Schroeder, Elias P. Duarte Jr.

Universidade Federal do Paraná – Departamento de Informática
Caixa Postal 19081 CEP 81531-990 – Curitiba - PR - Brasil

{jonatan,elias}@inf.ufpr.br

Abstract. *Routing protocols present a latency, i.e. a time interval spent to update all routing tables, after the topology changes. During the convergence latency interval several packets and connections may be lost. This work proposes a new strategy for a dynamic routing, in which intermediate routers, that may hold more recent information about the current state of the topology, select an alternative path when the regular one is not working. This routing strategy is based on strongly connected paths, chosen using connectivity criteria. Experimental results, based on simulation, comparing this approach and one using Dijkstra's algorithm, on a faulty environment, show that path changes made on strongly connected paths are 40% shorter than those found by Dijkstra's algorithm.*

Resumo. *Protocolos de roteamento apresentam uma latência, isto é, um intervalo de tempo para atualizar suas tabelas de rotas em toda a rede, após uma alteração na topologia. Esta latência de convergência pode gerar potenciais perdas de pacotes ou conexões na rede. Este trabalho propõe uma estratégia para roteamento dinâmico, permitindo que roteadores intermediários que possuam informações mais recentes de alterações na topologia interfiram no caminho utilizado. Este roteamento é baseado em caminhos robustos, escolhidos utilizando critérios de conectividade, que valorizam a redundância de caminhos. Uma comparação de resultados experimentais obtidos através de simulação da abordagem baseada nestes caminhos com uma seleção de caminhos por Dijkstra, em um ambiente sujeito a falhas, demonstra que as alterações de caminho feitas em caminhos robustos são 40% menores que as encontradas por Dijkstra.*

1. Introdução

Os protocolos de roteamento na Internet necessitam de um certo tempo para atualizar as tabelas de rotas de todos os roteadores, de modo a refletir mudanças de estado que tenham ocorrido na topologia da rede. Este período de tempo é conhecido como *latência de convergência* do protocolo [10]. A latência média do protocolo BGP (*Border Gateway Protocol*), um dos mais utilizados na Internet atualmente [12], é de 3 minutos, sendo que já foram observados períodos de latência de até 15 minutos [10].

Durante o período de latência do protocolo, alguns roteadores possuem informações inconsistentes nas suas tabelas de rotas, gerando potenciais perdas de pacotes e de conexões entre as aplicações que se comunicam pela rede.

Diversas abordagens têm sido propostas para resolução deste problema. Chandrashekar et.al. [1] propõem um mecanismo para análise das dependências de caminhos, de forma a reduzir a exploração de novos caminhos e, por conseqüência, reduzir a latência

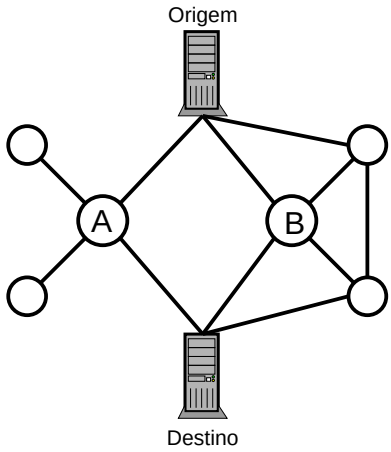


Figura 1: Exemplo de vantagem do uso da conectividade.

de convergência. Pei et.al. [11] propõem uma série de asserções a serem aplicadas nas redes, com o propósito de comparar caminhos similares e eliminar caminhos inviáveis. Wang et. al. [16] citam uma estratégia baseada em um mecanismo escalável, denominado FRTR (*Fast Routing Table Recovery*), para detecção e recuperação de inconsistências nas tabelas de roteamento do protocolo BGP.

Grande parte destas abordagens busca reduzir a latência de convergência, sem, no entanto, propor soluções a serem adotadas durante esta convergência. O que se propõe é reduzir os impactos da alta latência de convergência pela própria redução desta latência.

Outra abordagem para contornar esta situação [3, 13] se baseia na identificação de nodos altamente conexos na rede, encontrando-se caminhos que passem por estes nodos. Propõe-se que, em caso de ocorrência de falhas na conexão, se encontre um desvio que passa pelo nodo altamente conexo de forma a chegar ao destino. Esta abordagem pode ser visualizada na figura 1. Nesta figura, pode-se utilizar dois caminhos, passando um pelo nodo A e outro pelo nodo B. O caminho que passa por B pode ser considerado melhor, visto que há uma possibilidade maior da utilização de desvios, caso o caminho principal falhe.

Este trabalho propõe uma abordagem de roteamento tolerante a falhas e dinâmico. O roteamento é dinâmico no sentido que roteadores intermediários do caminho escolhido podem alterar este caminho utilizando informações sobre seu estado. Por exemplo, se um determinado roteador identifica que, no caminho escolhido para a comunicação, há um enlace falho, ele evita este caminho, e as conseqüentes perdas de pacote. Isto ocorre porque, após uma falha ou recuperação de um enlace, os roteadores mais próximos a este enlace recebem a informação deste evento antes dos demais pontos da rede. Este conceito pode se aplicar também a informações de congestionamento. O caminho alternativo encontrado após identificação da falha é chamado *desvio*.

Para que a possibilidade de determinar desvios sem falha e curtos seja maior, é necessário escolher caminhos com grande redundância, que chamaremos de caminhos robustos. Este trabalho propõe uma nova abordagem para a identificação de caminhos robustos, baseada em critérios de conectividade. Nesta abordagem, a conectividade média de *todos* os nodos do caminho é considerada na escolha dos caminhos. Em caminhos com maior conectividade, a possibilidade de encontrar desvios aumenta. Isto se deve à redundância de caminhos introduzida naturalmente pela alta conectividade.

Um dos principais critérios de conectividade utilizados neste trabalho é o número de conectividade, denotado por $\#C(v)$ e originalmente introduzido em [3]. Este número

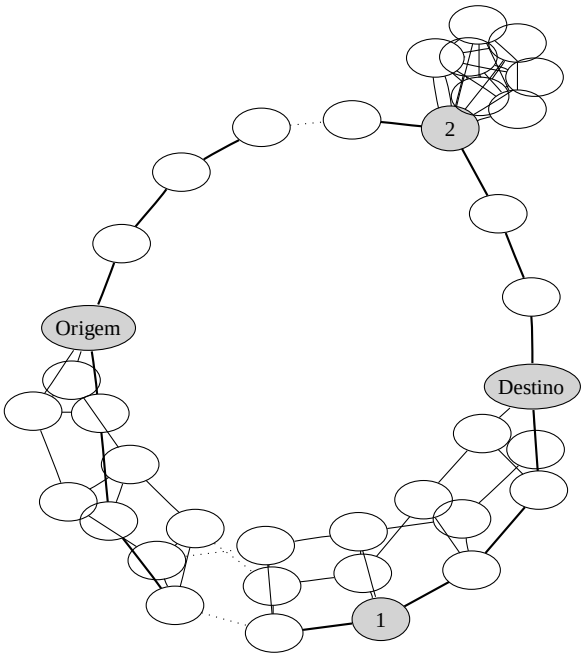


Figura 2: Exemplo de vantagem do uso da conectividade média.

é associado a um nodo específico, e corresponde à quantidade de caminhos disjuntos existentes entre o nodo em questão e seus vizinhos. Prova-se que o número de conectividade de um nodo está diretamente ligado à quantidade de caminhos alternativos que passam por esse nodo.

A figura 2 ilustra a vantagem do uso da conectividade média em relação à seleção de componentes de alta conectividade. No caminho que passa pelo nodo "1", todos os nodos possuem uma conectividade relativamente alta, isto é, $\#C(v) = 4$, em média. No outro caminho, o nodo "2" possui uma conectividade extremamente alta, com $\#C(v) = 8$, mas os demais nodos do caminho possuem uma conectividade baixa, $\#C(v) = 2$, reduzindo-se a possibilidade de desvios, dada uma falha no caminho original.

Deve ser destacado que este trabalho, em comparação com [3], apresenta duas contribuições principais: enquanto que em [3] a escolha do caminho é feita considerando apenas um nodo altamente conexo, este trabalho apresenta uma seleção baseado na conectividade média de todos os nodos do caminho; além disso, este trabalho apresenta uma proposta de roteamento dinâmico, no qual o caminho é alterado à medida em que informações mais atuais da topologia são descobertas.

É apresentada também uma avaliação baseada na comparação quantitativa entre o roteamento com a escolha do menor caminho pelo algoritmo de Dijkstra [2], e a nova abordagem apresentada. Esta comparação tem como objetivo avaliar a robustez e o custo do caminho, no que se refere à possibilidade de utilização de desvios para chegar ao destino, em caso de ocorrência de falhas no caminho original.

Este artigo está organizado como segue. Na seção 2 são descritos os principais critérios de conectividade utilizados neste trabalho. Na seção 3 é descrito o algoritmo utilizado para a seleção de caminhos robustos. Na seqüência, a seção 4 descreve a abordagem de roteamento tolerante a falhas utilizada, bem como o re-roteamento adotado em caso de identificação de falhas. A seção 5 a avaliação da confiabilidade dos caminhos encontrados, bem como apresenta os resultados experimentais obtidos. Por fim, a seção 6 apresenta as conclusões.

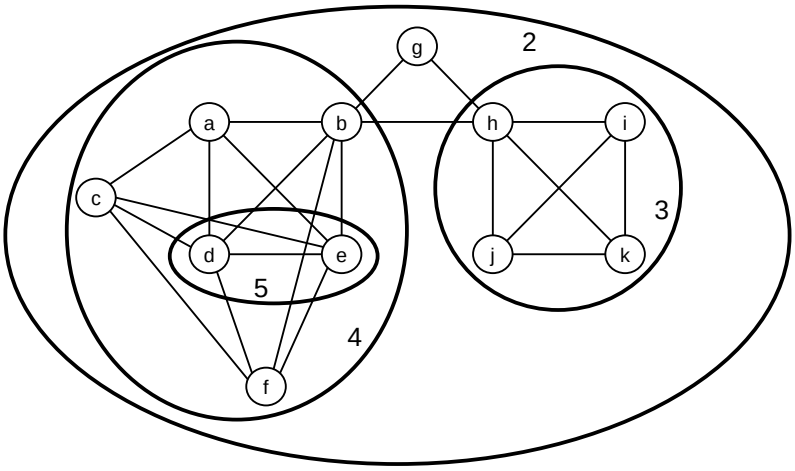


Figura 3: Um exemplo de cálculo do $\#C(v)$ e do $MCC(v)$.

2. Critérios de Conectividade

A seguir, são apresentados os critérios de conectividade utilizados neste trabalho, introduzidos originalmente em [3].

Em todas as definições, a topologia física (ou lógica) da rede é representada através de um grafo não-direcionado. Um grafo não direcionado é um par $G = (V, E)$, onde V é um conjunto (finito) de elementos denominados nodos (ou vértices) e E é um conjunto (finito) de elementos denominados arestas. Cada aresta é um conjunto $e = \{u, v\}$, onde $u, v \in V$.

Definição 2.1: *Dados dois vértices $u, v \in V$. Um corte entre esses dois vértices é um conjunto de arestas tais que, removendo-se as arestas do grafo original, os vértices u e v ficam desconexos no grafo G. Um corte mínimo entre dois vértices é um corte entre esses vértices com cardinalidade (ou seja, número de arestas) mínima.*

Definição 2.2: *Dado um vértice $v \in V$. Dado um subconjunto $V' \subseteq V$, com $v \in V'$, tal que a cardinalidade de qualquer corte mínimo em G que desconecta dois vértices em V' é máxima. O número de conectividade $\#C(v)$ é definido como a cardinalidade do corte mínimo entre qualquer par de vértices do subconjunto V'.*

Definição 2.3: *Dado um vértice $v \in V$. O componente de conectividade máxima $MCC(v)$ é o maior subconjunto de V contendo v tal que todo corte entre qualquer par de vértices do subconjunto possui cardinalidade maior ou igual a $\#C(v)$.*

A figura 3 ilustra um grafo juntamente com os valores dos números de conectividade e dos componentes de conectividade máxima desses nodos. Os círculos representam o $MCC(v)$ dos nodos em questão, juntamente com o número de conectividade associado. O número de conectividade de cada nodo corresponde ao número associado ao círculo mais restrito que corresponde ao mesmo. Por exemplo, considere o vértice a. O corte mínimo entre este vértice e o vértice b é 4, visto que é necessário remover 4 arestas para que ambos fiquem desconexos entre si. O mesmo ocorre com os vértices c, d, e e f. Desta forma, temos $\#C(a) = 4$ e $MCC(a) = \{a, b, c, d, e, f\}$.

2.1. Algoritmos de Obtenção dos Critérios de Conectividade

Uma propriedade importante do número de conectividade é a de que, dado um nodo $v \in V$, o valor do $\#C(v)$ é igual à cardinalidade máxima entre todos os cortes mínimos separando v de qualquer outro nodo de G. A prova pode ser verificada em [3].

Tendo em vista esta propriedade, utiliza-se a chamada árvore de corte, também conhecida por árvore de Gomory-Hu [5], para o cálculo dos critérios de conectividade.

Esta árvore tem algumas propriedades que auxiliam em cálculos relacionados ao corte mínimo em grafos não-direcionados. Diversos trabalhos, dentre os quais o trabalho de Gusfield [6], descrevem algoritmos para obtenção de uma árvore de corte.

Uma *árvore de corte* de um grafo $G = (V, E)$ é definida como uma árvore $T = (V_T, E_T, w)$, onde V_T é o conjunto de nodos da árvore, E_T é o conjunto de arestas da árvore, e $w : W \rightarrow \Re$ é uma função que determina o peso de uma aresta. Além disso, ela deve possuir as seguintes propriedades:

1. $V_T = V$, ou seja, os nodos da árvore de corte correspondem aos nodos do grafo G;
2. A cardinalidade de um corte mínimo entre dois vértices de G é dada pelo valor da aresta de menor peso pertencente ao caminho único que liga os dois vértices correspondentes em T;
3. Um corte mínimo entre dois vértices de G é obtido removendo a aresta mencionada acima e considerando os dois conjuntos de vértices induzidos pelas duas sub-árvores formadas pela remoção de tal aresta em T.

A partir da árvore de corte, é possível calcular os critérios de conectividade citados no início desta seção. Em [3] prova-se que, para um grafo G, uma árvore de corte T de G e um vértice v, o valor do $\#C(v)$ é igual ao valor da aresta de maior peso incidente a v em T. No mesmo trabalho também se prova que os vértices alcançáveis em T a partir de v, utilizando apenas arestas com peso igual ou superior ao valor do $\#C(v)$, pertencem ao conjunto $MCC(v)$.

Para obtenção da árvore de corte, foi utilizado o algoritmo de Gusfield [6]. Neste algoritmo, para obtenção do corte mínimo, foi utilizada uma simplificação do algoritmo de Ford e Fulkerson [4] para cálculo de fluxo máximo/corte mínimo. Esta simplificação é descrita em [14].

3. Cálculo do Caminho

Este trabalho propõe uma nova abordagem para roteamento baseada em critérios de conectividade. Para a seleção de um caminho, é considerada a conectividade média de todos os seus nodos; outros critérios como o tamanho do caminho também são considerados. O algoritmo para a avaliação e a seleção de caminhos é descrito a seguir.

3.1. Seleção dos Caminhos

O nosso objetivo, nesta seção, é o de identificar os caminhos mais robustos entre dois pontos de uma rede. Considerando que a rede em que o cálculo será efetuado está modelada em um grafo não-direcionado G, vamos selecionar os melhores caminhos entre dois vértices: uma origem s, e um destino t.

Inicialmente é realizada uma busca em profundidade no grafo G, partindo da origem s. Esta busca retorna todos os caminhos p sem ciclos entre s e t. Após esta busca, para cada um dos caminhos encontrados é atribuído um valor $\Gamma(p)$, que é obtido considerando os critérios de conectividade dos nodos do caminho, e o tamanho do caminho em termos de seu número de arestas. O valor final associado a cada caminho é dado pela seguinte equação:

$$\Gamma(p) = \omega_1.E\big[\#C(v)\big] + \omega_2.E\big[\,|MCC(v)|\,\big] + \omega_3.E\big[\,|p_i|\,\big] \qquad (1)$$

Nesta equação, são utilizados os seguintes componentes:

- $E\big[\#C(v)\big]$: média do número de conectividade ($\#C(v)$) de todos os nodos intermediários do caminho p (exceto s e t);
- $E\big[\,|MCC(v)|\,\big]$: média do tamanho do componente de conectividade máxima ($MCC(v)$) de todos os nodos intermediários do caminho p (exceto s e t);
- $E\big[\,|p|\,\big]$: tamanho do caminho p, em termos do número de arestas atravessadas pelo caminho.

Cada um dos componentes deste cálculo é multiplicado por um peso ω_i, de forma a possibilitar dar mais ênfase a um aspecto ou outro. Diversas combinações de pesos são possíveis, tendo em vista requisitos específicos de cada sistema. Para o tamanho do caminho, recomenda-se o uso de pesos negativos ($\omega_3 < 0$), visto que, quanto menor o caminho, mais adequado ele é. Para a conectividade média, é recomendado que o peso seja positivo ($\omega_1 > 0$), uma vez que o objetivo deste algoritmo é encontrar caminhos robustos, e a conectividade é fator determinante da robustez do caminho. O peso do tamanho do componente de conectividade máxima também deverá ser positivo ou nulo ($\omega_2 \geq 0$), pois possui relação direta com a robustez, porém recomenda-se que seja relativamente menor que o peso da conectividade, uma vez que nodos com conectividade baixa normalmente pertencem a um componente de conectividade máxima maior, afetando o resultado esperado da robustez. Este fato é ilustrado na figura 3. Nesta figura, o nodo g possui baixa conectividade, mas o $MCC(g)$ corresponde ao grafo por inteiro. Já o nodo d possui alta conectividade, porém o $MCC(d)$ contém apenas os nodos d e e.

Há também a possibilidade de trabalhar com uma variabilidade no peso específico para o cálculo da média do número de conectividade. Desta forma, em lugar de utilizar uma média simples, utiliza-se a seguinte fórmula:

$$E\big[\#C(v)\big] = \frac{\displaystyle\sum_{k=1}^{|p|-1} \alpha^{k-1} \#C(n_k)}{\displaystyle\sum_{k=1}^{|p|-1} \alpha^{k-1}} \tag{2}$$

Nesta equação, n_k são os nodos intermediários do caminho p, e α é o coeficiente de variação da média. Desta forma, se $0 < \alpha < 1$, teremos uma conectividade média com ênfase na conectividade dos nodos mais próximos à origem. Se $\alpha > 1$, a média da conectividade será mais afetada pelos nodos mais próximos ao destino. Esta situação, a princípio, será mais favorável, uma vez que a necessidade de que desvios sejam encontrados no final do caminho é maior que no início do caminho. Considerando $\alpha = 1$, a média da conectividade será linear, com todos os nodos sendo considerados igualmente.

O mesmo tratamento pode ser feito com o tamanho do componente de conectividade máxima, de acordo com a seguinte equação:

$$E\big[\,|MCC(v)|\,\big] = \frac{\displaystyle\sum_{k=1}^{|p|-1} \beta^{k-1} \,|MCC(n_k)|}{\displaystyle\sum_{k=1}^{|p|-1} \beta^{k-1}} \tag{3}$$

Da mesma forma que o resultado da média da conectividade é afetado pelo valor de α, a média do tamanho do componente de conectividade máxima é afetado pelo valor de β. Valores de β menores ou maiores que 1 darão mais ênfase aos nodos mais próximos à origem ou ao destino, respectivamente. Para β igual a 1, a média será linear.

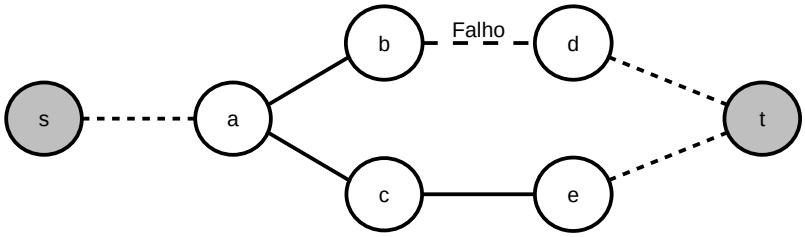


Figura 4: Exemplo da utilização de um desvio.

Após a atribuição do valor descrito acima, os caminhos são ordenados pelo valor $\Gamma(p)$ de forma decrescente. Assim, o caminho com maior valor é considerado o mais robusto, e deve ser a primeira escolha para o roteamento. Em caso de empate, foi considerado critério de desempate o tamanho do caminho. Caso dois caminhos possuam mesma avaliação e mesmo tamanho, um dos dois caminhos é selecionado aleatoriamente.

No caso em que há uma aresta ligando diretamente a origem e o destino, não há nodos intermediários para o cálculo da conectividade média. Podemos assumir, neste caso, que a aresta será o caminho mais apropriado. Desta forma, caso haja um caminho direto, com apenas uma aresta, entre a origem e o destino, este é considerado o primeiro caminho, sem que seja necessária avaliação do mesmo. Vamos considerar que, neste caso, $\Gamma(p) = +\infty$.

4. Uma Proposta de Roteamento Tolerante a Falhas

O principal objetivo do roteamento confiável proposto neste trabalho é a escolha de caminhos robustos, em que, quando uma falha é identificada, um desvio possa ser encontrado, ou seja, um novo caminho que chegue ao destino a partir da origem na presença da falha. Nesta seção descreve-se uma proposta de roteamento baseado nesta identificação de desvios.

O roteamento proposto neste trabalho se baseia na permissão para que roteadores intermediários de um caminho escolhido interfiram no restante do caminho. Esta permissão é concedida porque as informações de topologia normalmente são atualizadas mais rapidamente em nodos mais próximos aos eventos que geram sua alteração. Por exemplo, a ocorrência de uma falha em um enlace é primeiramente repassada para os nodos vizinhos, que por sua vez passam para seus vizinhos e assim sucessivamente. Tendo em vista este fato, roteadores que tenham o conhecimento de uma falha na seqüência do caminho proposto podem alterar o caminho para outro caminho que não esteja falho. Este novo caminho é denominado desvio.

A figura 4 ilustra uma situação em que a utilização de desvios é demonstrada. Supondo uma conexão entre os nodos s e t, em que s deseja enviar um pacote de dados para t. Vamos supor que o caminho escolhido seja o caminho que passa pelos nodos a, b e d. O procedimento para escolha do caminho está sendo ignorado, por enquanto.

Considere a situação em que a aresta que liga os nodos b e d falha. Se não for utilizada uma abordagem tolerante a falhas, o pacote enviado pelo nodo s será simplesmente perdido. Considerando a comunicação confiável, quando o nodo s identificar esta falha, seja por *timeout*, seja por informação por parte dos outros nodos, ele enviará um novo pacote. Este pacote possivelmente seguirá o mesmo caminho utilizado pelo pacote anterior, visto que o nodo s pode não ter sido informado da falha na aresta em questão. Este procedimento poderá se repetir até que s possua a informação mais recente da topologia, o que pode demorar até 15 minutos na Internet com o protocolo BGP, conforme já mencionado em seções anteriores.

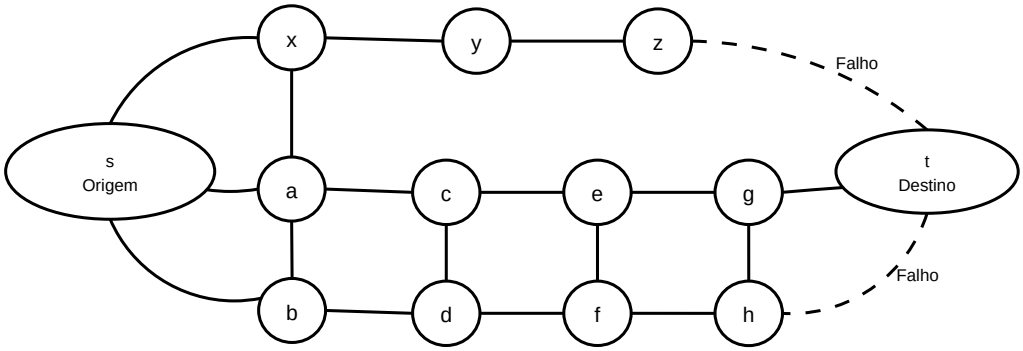


Figura 5: Exemplo de grafo com falhas.

Pela abordagem tolerante a falhas proposta neste trabalho, ao chegar no nodo b, a falha será identificada, e o nodo b fica, então, encarregado de escolher um novo caminho chegando em t, já desconsiderando o enlace entre os nodos b e d. Como b não possui outro enlace que chegue ao nodo t, é necessário retornar ao nodo a. Porém, para que a não tente enviar o pacote novamente por b, é necessário informar ao nodo a que o enlace $b - d$ está falho. Após a atualização das informações da topologia, a escolhe um novo caminho até t, que será, por exemplo, o caminho que passa por c e e.

Para que esta abordagem produza resultados satisfatórios, é necessário que haja uma redundância nos caminhos entre os roteadores intermediários e o nodo de destino. Esta redundância é necessária para que haja desvios que não precisem retornar até a origem para alcançar o destino. Desta forma, conforme já descrito nas seções anteriores, torna-se importante a escolha de caminhos robustos baseada em critérios de conectividade, que introduzem naturalmente a redundância dos caminhos.

5. Avaliação de Confiabilidade

Conforme já descrito nas seções anteriores, o objetivo do roteamento confiável baseado na escolha de caminhos robustos é possibilitar que, em caso de identificação de uma falha no caminho, seja possível encontrar um desvio, ou seja, um novo caminho que chegue ao destino sem que seja necessário retornar ao nodo inicial.

Para avaliar se o objetivo foi encontrado, este trabalho realiza uma comparação com o algoritmo de Dijkstra [2]. O objeto de avaliação é a robustez do caminho, ou seja, a possibilidade de encontrar um desvio em caso de falha. A métrica utilizada neste trabalho para comparação quantitativa da robustez de um caminho é o tamanho do caminho total atravessado pelos dois algoritmos. Este caminho inclui as arestas atravessadas até o nodo adjacente a uma aresta falha, e as arestas do novo caminho (desvio) entre esse nodo e o destino. O desvio foi calculado da mesma forma que o caminho original, porém desconsiderando a aresta identificada como falha e tendo como origem o nodo adjacente a esta aresta.

Na figura 5 podemos observar como ambos os algoritmos se comportam em relação a desvios. Pelo algoritmo de Dijkstra, o caminho escolhido será o caminho $p_D = (s, x, y, z, t)$. Porém, ao chegar ao nodo z, identifica-se uma falha que impossibilita a continuidade do caminho. Desta forma, um novo caminho é escolhido para chegar de z até o destino t. Tendo em vista que a aresta (z, t) já foi identificada como falha, ela não poderá ser utilizada, e o melhor caminho então é $p'_D = (z, y, x, a, c, e, g, t)$. Desta forma, o caminho total percorrido é de 10 arestas, 3 de s até z e 7 de z até t.

Pela abordagem descrita neste trabalho se utilizarmos como pesos os valores 2, 0.001 e -1 respectivamente para ω_1, ω_2 e ω_3, e com α e β valendo 1, os melhores caminhos

serão $p_S = (s, a, c, e, g, t)$ e $p'_S = (s, b, d, f, h, t)$, que ficarão com mesma avaliação ($\Gamma(p_S) = \Gamma(p'_S)$). Para fins de entendimento do funcionamento do algoritmo, suponhamos que o caminho escolhido é p'_S. Percorrendo este caminho, ao chegar em h, é identificada uma falha, que impossibilita a continuidade do caminho. É necessário, então, encontrar um novo caminho entre h e t que não contenha esta aresta. Chegamos facilmente ao caminho $p''_S = (h, g, t)$. Se verificarmos o caminho total, podemos verificar que foram percorridas 6 arestas, 4 de s até h e 2 de h até t.

O exemplo acima ilustra a robustez dos caminhos que estão sendo propostos neste trabalho. Porém, para uma avaliação mais completa, foi desenvolvida uma plataforma de simulação, de forma a verificar o comportamento dos algoritmos em situações diversas. Esta plataforma recebe como entrada um grafo $G = \langle V, E \rangle$ correspondente à topologia de uma rede, e realiza testes de robustez neste grafo.

A plataforma descrita acima considera, para o cálculo, todas as situações de falhas possíveis em que haja uma aresta falha. Para cada aresta, são considerados todos os pares possíveis de origem e destino na rede, em que a origem e o destino sejam nodos diferentes do grafo G. Temos, portanto, uma situação σ descrita da seguinte forma:

$$\sigma = \langle e, s, t \rangle, e \in E(G), s, t \in V(G), s \neq t$$

Para cada situação σ, o caminho p_D entre os nodos s e t é calculado pelo algoritmo de Dijkstra. Igualmente, o caminho p_S é calculado pelo algoritmo proposto neste trabalho. Em ambos os caminhos, verifica-se se a aresta e pertence ao caminho. Se pertencer, é calculado um novo caminho p'_D (ou p'_S) entre o nodo em que a falha é identificada e o nodo t, agora desconsiderando a aresta e. Neste caso, o tamanho total considerado é a soma entre o caminho até o ponto em que a falha ocorre e o tamanho do caminho p'_D (ou p'_S). Caso a aresta e não pertença ao caminho, o tamanho total considerado é o próprio tamanho do caminho p_D (ou p_S).

Os valores encontrados são armazenados, e após a avaliação de todas as situações, a média dos valores encontrados para o algoritmo de Dijkstra e para o algoritmo proposto é apresentado.

5.1. Implementação

De forma a possibilitar a avaliação da robustez de caminhos, foi implementada uma ferramenta de cálculo e avaliação dos conceitos apresentados neste trabalho. Esta implementação foi feita na linguagem Java 1.4.2 [8].

Foi utilizada uma biblioteca disponível na Internet para trabalhar com grafos, denominada JDigraph [9]. Além disso, foi utilizada a estrutura de Servlets e JSPs (*Java Server Pages*), introduzida pela plataforma J2EE (*Java 2 Enterprise Edition*) [7]. Para executar estas estruturas, foi utilizado o servidor Java Tomcat [15].

Foi criada uma interface gráfica de acesso aos cálculos, acessível através de um browser da Web. Esta interface é demonstrada na figura 6. Nesta figura, (A) ilustra a página inicial, em que é solicitado o arquivo de grafo ou a estrutura para criação manual. (C) mostra a página principal, que lista os critérios de conectividade encontrados, bem como permite a seleção de nodos para cálculo dos melhores caminhos. Estes caminhos são mostrados numa nova página, ilustrada em (B). A partir da página principal, também é possível realizar uma simulação completa do grafo, envolvendo uma avaliação de todos os caminhos possíveis, com todas as situações de falha possíveis, conforme descrito anteriormente nesta seção. O resultado desta avaliação é demonstrado na mesma figura, em (D).

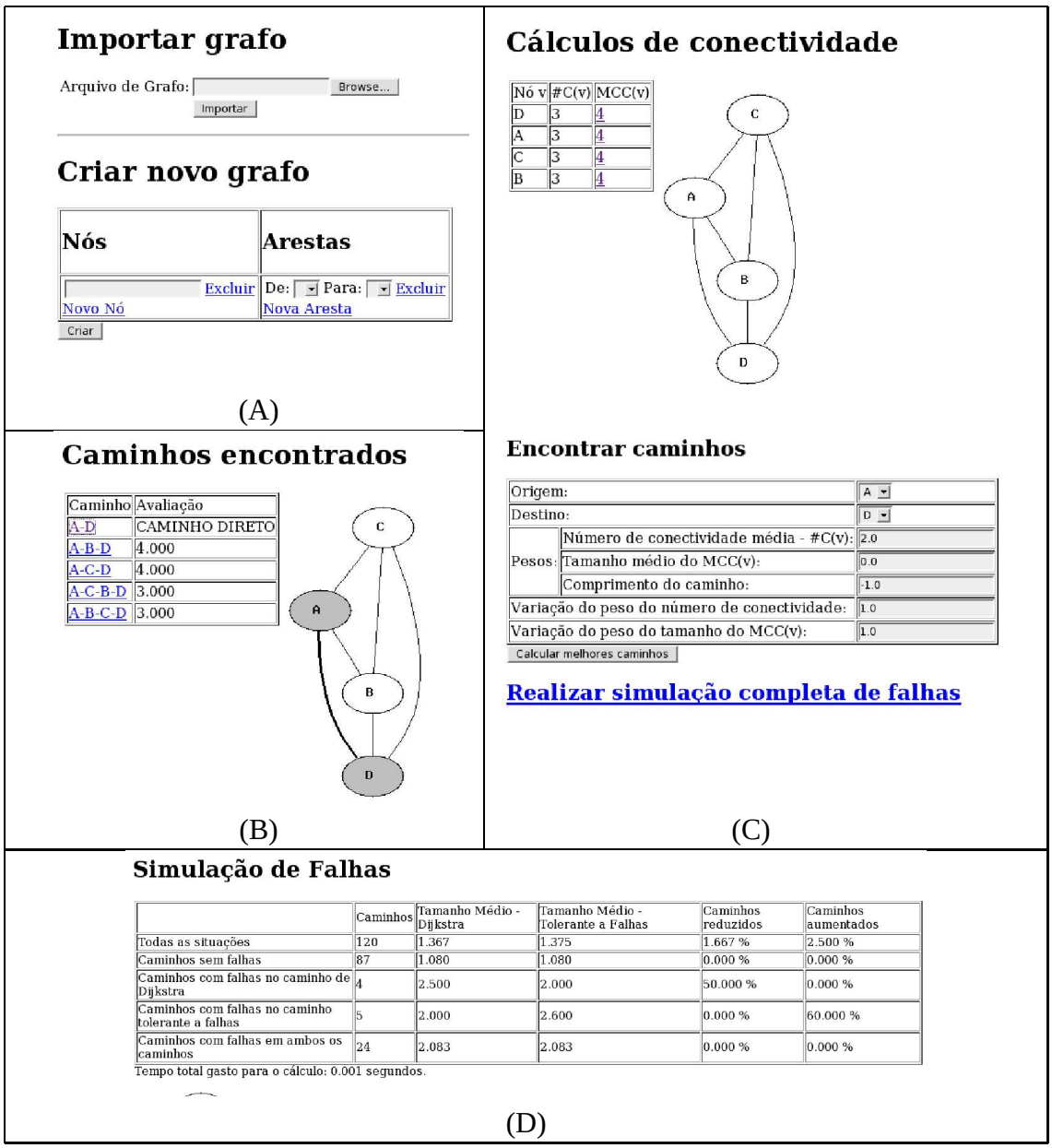


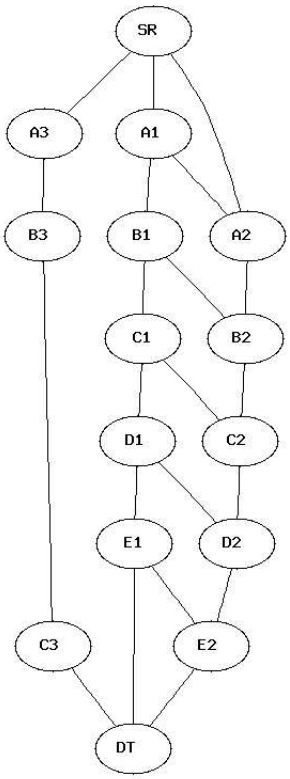
Figura 6: Exemplo da interface gráfica.

Um exemplo de grafo utilizado para testes é o mostrado na figura 7. Foi realizado um teste de simulação completa de falhas, conforme descrito no início desta seção. Os testes foram realizados com os parâmetros ω_1, ω_2, ω_3, α e β, descritos na seção 3, valendo, respectivamente, 2, 0.001, -1, 1 e 1.

O resultado da simulação de falhas pode ser visto na figura 7. Nesta figura, pode-se verificar que, em situações em que falha interrompe apenas o caminho de Dijkstra, o tamanho médio do caminho pela nova abordagem é 2,2 arestas menor que o caminho de Dijkstra, que necessitou do desvio. Porém, quando a falha interrompeu apenas o caminho da nova abordagem, a diferença entre os caminhos passou para 1,3, demonstrando que o desvio gerado neste caso é menor que o desvio pelo caminho de Dijkstra.

6. Conclusão

Este trabalho propôs uma estratégia de roteamento dinâmico e tolerante a falhas, que se baseia na escolha de caminhos robustos e na utilização de caminhos alternativos (desvios) em caso de falhas no caminho inicial utilizado. O roteamento proposto produziu



	Número de Caminhos	Tamanho Médio	
		Dijkstra	Robusto
Caminhos sem falhas	7371	2.664	2.673
Caminho de Dikstra com falha	261	6.418	4.261
Caminho robusto com falha	269	4.186	5.494
Caminho de ambos os tipos com falha	919	5.751	5.766
Todos os caminhos (média geral)	8820	3.143	3.128

Figura 7: Resultados obtidos com um grafo de exemplo.

resultados preliminares satisfatórios no que se refere à identificação de desvios. Os caminhos alternativos são relativamente curtos, tendo em vista que não há necessidade de retornar ao nodo de origem do pacote. Os caminhos robustos podem caracterizar uma possibilidade maior de desvios, dada a redundância introduzida por estes caminhos.

Trabalhos futuros devem incluir testes mais abrangentes, com grafos maiores e utilizando métodos como o de Waxman [17] para geração dos grafos, além do uso de valores distintos para os parâmetros ω_1, ω_2, ω_3, α e β. Também será estudada a utilização de técnicas e heurísticas para reduzir o tempo de escolha do caminho, como o descarte de alguns caminhos e o cálculo heurístico do $\widetilde{\#C}(v)$ proposto em [3]. Além disso, devem ser examinados aspectos relacionados ao roteamento dinâmico, além da realização de testes com situações de mais de uma falha, bem como com falhas de nodos em lugar de apenas de arestas. Também serão avaliados outros critérios para inclusão na avaliação dos caminhos, como a latência e o congestionamento do caminho, assim como medidas de dispersão dos critérios de conectividade, como o desvio padrão. Futuramente deve-se avaliar formas de avaliação de caminhos em redes com topologia parcialmente desconhecida, como no roteamento entre sistemas autônomos distintos.

Referências

[1] J. Chandrashekar, Z. Duan, Z. L. Zhang, and J. Krasky. Limiting path exploration in BGP. disponível em `http://www-users.cs.umn.edu/~jaideepc/papers/epic-tr.pdf`.

[2] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill, second edition, 1990.

[3] E. P. Duarte Jr., R. Santini, and J. Cohen. Delivering packets during the routing convergence latency interval through highly connected detours. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'2004)*, pages 495–504, Florence, Italy, 2004.

[4] L. R. Ford Jr. and D. R. Fulkerson. Flows in networks. *Princeton University Press*, 1962.

[5] R. E. Gomory and T. C. Hu. Multi-terminal network flows. In *SIAM Journal on Applied Mathematics*, volume 9, pages 551–556, 1961.

[6] D. Gusfield. Very simple method for all pairs network flow analisys. In *SIAM Journal on Computing*, volume 19(1), pages 143–155, 1990.

[7] Java 2 Platform, Enterprise Edition (J2EE). `http://java.sun.com/j2ee`.

[8] Java Technology. `http://java.sun.com`.

[9] JDigraph. `http://jdigraph.dev.java.net`.

[10] C. Labovitz, A. Ahuja, A. Bose, and F. Jahanian. Delayed internet routing convergence. In *SIGCOMM*, pages 175–187, 2000.

[11] D. Pei, X. Zhao, L. Wang, D. Massey, A. Mankin, S. Wu, and L. Zhang. Improving BGP convergence through consistency assertions. In *INFOCOM*, New York, 2002.

[12] Y. Rekhter and T. Li. *RFC 1771: A Border Gateway Protocol 4 (BGP-4)*, March 1995.

[13] R. Santini, E. P. Duarte Jr., J. Schroeder, P. R. Torres Jr., and J. Cohen. Roteamento tolerante a falhas baseado em desvios de alta conectividade. Technical report, Universidade Federal do Paraná, 2004.

[14] J. Schroeder, A. L. P. Guedes, and E. P. Duarte Jr. Computing the minimum cut and maximum flow of undirected graphs. Technical report, Universidade Federal do Paraná, 2004.

[15] The Jakarta Site - Apache Jakarta Tomcat. `http://jakarta.apache.org/tomcat`.

[16] L. Wang, D. Massey, K. Patel, and L. Zhang. FRTR: A scalable mechanism for global routing table consistency. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'2004)*, pages 465–474, Florence, Italy, 2004.

[17] B. M. Waxman. Routing of multipoint connections. In *IEEE Journal of Selected Areas in Communications*, volume 6(9), pages 1617–1622, 1988.

Avaliação de um Mecanismo de Checkpointing para o MyGrid

Jeysonn Isaac Balbinot, Ingrid Jansch-Pôrto,
Hélio Miranda Silva, Taisy Silva Weber

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

{jibalbinot,ingrid,hamsilva,taisy}@inf.ufrgs.br

Abstract. *The heterogeneity, complexity and software and hardware diversity found in grids make the probability of failures higher than in traditional distributed systems. MyGrid front-end is a middleware that gives support for the execution of applications organized as Bag-of-Tasks in grid environments. A central node controls the application execution. To avoid the loss of computation in case of crash of this central node, which is a single-point-of failure, a rollback-recovery mechanism was proposed to work in conjoint with the scheduler. This paper uses SimGrid toolkit to evaluate, by simulation, the impact of this mechanism over the performance of the applications.*

Resumo. *A heterogeneidade, complexidade e diversidade de software e hardware dos grids tornam a probabilidade de defeitos maior que em sistemas distribuídos tradicionais. O MyGrid é um middleware que dá suporte à execução de aplicações do tipo Bag-of-Tasks em um grid. O controle das aplicações é feito por um nodo central que se torna ponto único de falha. Para evitar a perda de computação no caso de colapso deste, foi proposto um mecanismo de recuperação por retorno vinculado ao código do escalonador. Este trabalho avalia, via simulação, o impacto deste mecanismo no desempenho das aplicações, utilizando a ferramenta SimGrid.*

1. Introdução

O crescente avanço do *hardware* e *software*, bem como o aumento da velocidade e qualidade dos serviços de rede oferecidos atualmente, mostram-se como uma alternativa para a exploração do poder computacional. O uso de uma grande quantidade de máquinas interligadas, com fraco acoplamento, dá origem a um novo conceito, conhecido como computação em *grid*[1].

O *grid* constitui-se em uma plataforma para o compartilhamento coordenado de recursos distribuídos geograficamente. Possui como objetivo resolver uma gama de problemas e aplicações distribuídas nas mais diversas áreas, como ciência, engenharia e comércio, atendendo assim a organizações virtuais multi-institucionais e dinâmicas. Os serviços oferecidos pelo *grid*, e.g. o acesso sob demanda aos recursos, devem ter baixo custo, oferecendo um certo grau de consistência e confiabilidade [6][7].

[1] Embora o termo *grid* venha sendo traduzido por alguns pelo vocábulo *grade*, em português, optou-se por mantê-lo em sua versão original.

Devido às características intrínsecas de uma plataforma *grid* como, por exemplo: heterogeneidade de *hardware* e *software*, grande variedade de recursos espalhados em diversos domínios, compartilhamento do ambiente por várias aplicações e possibilidade de executar aplicações que podem utilizar até milhares de nodos, a probabilidade de ocorrência de falhas e defeitos torna-se muito alta [8], de maneira que a ocorrência de falhas pode ser vista quase como a regra, e não mais como a exceção.

O MyGrid [4][5][9] é uma ferramenta que auxilia no desenvolvimento de aplicações paralelas, com apoio ao gerenciamento de recursos e tarefas que compõem uma plataforma *grid*. O MyGrid provê suporte a execução de aplicações que sigam o modelo *Bag-of-Tasks* (BoT) – são tarefas independentes entre si e que podem ser escalonadas e executadas em qualquer ordem e em qualquer recurso que esteja disponível no *grid*. A total independência entre as tarefas faz com que não exista comunicação entre elas durante a computação [3].

Algumas soluções de escalonamento oferecidas pelo MyGrid utilizam replicação de tarefas como alternativa para o aumento de desempenho, sem a necessidade de informações adicionais sobre o *grid* ou sobre a aplicação. Além da busca de melhoria de desempenho com o uso de replicação, esta técnica também auxilia na tolerância a falhas. Desta forma, são tolerados defeitos em recursos do *grid* que estejam processando uma das réplicas. Atualmente, não existe outro mecanismo específico para tolerância a falhas. Defeitos que afetem o nodo central, o qual coordena a aplicação, acarretam a perda de todas as tarefas e resultados daquela fase da computação.

Considerando então a probabilidade real de falhas nestes sistemas, decidiu-se trabalhar no suporte às atividades do *grid*, melhorando a capacidade de resposta do sistema à ocorrência de defeitos. O mecanismo escolhido é o uso de *checkpoints* relacionados às atividades do escalonador de tarefas, visando a posterior recuperação de estado caso a máquina onde está sendo executado o escalonador sofra defeito. Porém, embora a técnica permita que a aplicação possa ser recuperada após a ocorrência de defeitos, ela não deve degradar o desempenho do sistema, de maneira significativa.

Dessa forma, é importante avaliar o impacto que o salvamento de estado causará sobre o desempenho normal do MyGrid. Decidiu-se fazer esta avaliação através de simulação, em função das vantagens: necessidade de recursos virtuais, controlabilidade do ambiente e facilidade de repetição dos experimentos.

Portanto, o objetivo deste trabalho é avaliar o custo da abordagem proposta – sobreposição do *checkpointing* à operação do escalonador do MyGrid – com intenção de melhorar a resposta a defeitos em arquiteturas em *grid*. A degradação do desempenho durante a operação normal (livre de falhas) de aplicações executadas no MyGrid é avaliada com o uso de simulações realizadas com o SimGrid [1][2][14].

Este artigo está organizado como segue. Na seção 2, são detalhados o funcionamento do MyGrid, sua estrutura operacional, políticas de escalonamento, e o modelo de *checkpointing* e recuperação propostos. Na seção 3, é descrita a ferramenta de auxílio para a simulação de *grids*, SimGrid. Na seção 4, é apresentado o modelo de simulação criado e um estudo de caso. Na seção 5, são apresentados os resultados obtidos com os experimentos. As conclusões finais estão na seção 6.

2. MyGrid

MyGrid [9] é uma ferramenta que visa dar suporte à utilização de recursos em um *grid* computacional e visa contornar dificuldades de gerenciamento e de escalonamento de tarefas neste ambiente. É um *software* de código aberto, implementado em Java e em

scripts shell. Pode ser utilizado tanto em plataformas Windows quanto Linux; a última versão (atualmente é 2.2) está disponível.

O *grid* de um determinado usuário é formado por todas as máquinas às quais ele tem permissão de acesso: isto é extremamente interessante para aplicações do tipo BoT, devido ao fraco acoplamento das tarefas. A execução simultânea de aplicações é coordenada pelo nodo central que escalona as tarefas nas máquinas que executam cada aplicação. O nodo central é chamado de *home machine* e os demais nodos são as máquinas do *grid* (*grid machines*)[2]. Cada máquina do *grid* executa uma tarefa de cada vez. Em uma configuração típica, essas máquinas não compartilham o sistema de arquivos, nem possuem os mesmos *softwares* instalados. Toda a comunicação é iniciada e coordenada pela *home machine* a qual, além de conter os dados de entrada de cada aplicação, é responsável por fazer a coleta dos resultados da computação.

O objetivo do MyGrid é oferecer um sistema simples, completo e seguro para execução de aplicações BoT em *grids*. O esforço para uso do MyGrid deve ser mínimo e o mais próximo possível de uma solução pronta (*plug-and-play*), por isso, simples. Completo identifica a necessidade de cobrir todo o ciclo de uso de um sistema computacional, do desenvolvimento à execução. A segurança é atingida pela proteção a vulnerabilidades no ambiente computacional do usuário [5].

Cada máquina do *grid* deve oferecer um conjunto mínimo de serviços, definidos pelo MyGrid, que compõem a *Grid Machine Interface* [4]. Estes serviços são: disparo (*start-up*) de tarefas em máquinas do *grid* (execução remota); cancelamento de tarefas que estejam em execução; transferência de arquivos das máquinas do *grid* para a *home machine*; transferência de arquivos da *home machine* para as máquinas do *grid.*

A *home machine* possui um escalonador (denominado *Scheduler*), o qual recebe do usuário a descrição das tarefas a executar, escolhe onde cada tarefa será executada, submete e monitora cada uma delas segundo sua estratégia de escalonamento.

2.1. Escalonamento de tarefas

Na prática, não é trivial obter informações precisas sobre o ambiente e a aplicação, principalmente em ambientes amplamente dispersos como os *grids* computacionais [12]. Características do *grid* tais como heterogeneidade dos recursos, dinâmica natural das máquinas, carga, largura de banda, latência e topologia da rede de comunicação, devem ser consideradas durante o escalonamento, principalmente quando a aplicação utiliza grande quantidade de dados [10]. Como alternativa à dificuldade de obtenção de informações dinâmicas sobre os recursos e sobre o tempo de execução da aplicação, existem escalonadores de aplicações BoT que atingem bom desempenho utilizando replicação. Porém, o uso de tal heurística implica um consumo extra de recursos [12].

Em sua versão atual, o MyGrid oferece como estratégias de escalonamento os algoritmos *Workqueue*, *Workqueue with Replication* e o *Storage Affinity*.

A estratégia *Workqueue* (WQ) não necessita de nenhuma informação ou previsão de desempenho para fazer o escalonamento das tarefas, utiliza apenas informações necessárias, como nomes das tarefas a serem escalonadas e processadores disponíveis[3]

---

[2] Ao longo deste artigo, a denominação *home machine*, usada no MyGrid, será mantida. As demais máquinas, entretanto, serão referenciadas pelo termo traduzido (máquinas do *grid*, máquina remota (para enfatizar a execução de atividades externamente à *home machine*) ou como máquinas, apenas).

[3] Do ponto de vista do MyGrid, processador disponível é aquele que está conectado ao *grid* e ainda não recebeu qualquer tarefa.

para executarem estas. As tarefas são escolhidas de forma arbitrária e escalonadas aos processadores disponíveis. Após a tarefa ter sido completada, o processo envia o resultado ao escalonador, ficando assim apto a receber uma nova tarefa [10].

Uma variação da estratégia WQ é a *Workqueue with Replication* (WQR): há replicação de tarefas, quando um processador se torna disponível e não há mais tarefas pendentes para serem colocadas em execução. Então o WQR replica tarefas que ainda estão em execução até um número máximo definido pelo usuário ou enquanto houver processadores disponíveis. Esta replicação permite usar o primeiro resultado obtido; a execução das demais tarefas (réplicas ou original) que ainda não haviam produzido resultados é interrompida. Esta abordagem melhora o desempenho, se comparada ao WQ, pois a replicação aumenta a probabilidade de que pelo menos uma delas seja atribuída a um processador rápido ou com baixa carga de trabalho. Uma tarefa não replicada poderia ser inconvenientemente atribuída a um processador lento ou sobrecarregado por outras atividades e retardar a obtenção do resultado do *job* [10].

A estratégia *Storage Affinity* visa melhorar o desempenho de aplicações que processam grandes quantidades de dados (*Processors of Huge Data*, PHD) [13]. Para tanto, é usado um método de priorização de tarefas baseado em afinidade de armazenamento (*storage affinity*). O valor da afinidade entre uma tarefa e um site (*host*) ou máquina do *grid* determina quão próximo deste a tarefa está, ou seja, ela está associada à quantidade de *bytes* da entrada da tarefa que já está armazenada remotamente em uma determinada máquina do *grid*. Além de aproveitar a reutilização dos dados, o *Storage Afinitty* também efetua a mesma abordagem de replicação de tarefas utilizada no WQR [12].

2.2. Modelo de recuperação

Falhas que afetam isoladamente as máquinas do *grid* são menos críticas do que as que afetam a *home machine*, porque resultam na perda de apenas uma tarefa que esteja sendo executada nesse momento e não de todo o *job*. Além disso, esse problema pode ser parcialmente contornado com o uso de replicação de tarefas ou minimizado se os usuários dimensionarem tarefas de granulosidade reduzida. A introdução de recuperação vinculada às tarefas isoladas foi descartada porque sua implementação transparente (do ponto de vista dos programadores de aplicações) é bem mais complexa.

Na estrutura do MyGrid, a *home machine* é o elemento crítico na composição do sistema (“ponto único de falha”) pois, se uma falha a afetar, as aplicações dependentes dela poderão ser interrompidas ou mesmo perdidas. Além disso, é necessário lembrar que, em uma estrutura de *grid*, haverá várias máquinas atuando como *home*, uma para cada um dos diferentes usuários. Com o objetivo de melhorar a tolerância a falhas que afetem a *home machine* do MyGrid e, conseqüentemente o sistema como um todo, optou-se pelo uso do salvamento de *checkpoints* para uma posterior recuperação por retorno (*rollback-recovery*) [11].

Os *checkpoints* contêm informações que refletem o estado do sistema e processos envolvidos e, após a constatação do defeito, permitem retomar a execução a partir de um estado consistente prévio à ocorrência da falha. Com este mecanismo, a execução anterior na *home machine* pode ser recuperada e continuar a execução de um *job* no mesmo nodo ou em outro nodo disponível, evitando a repetição de execuções anteriores concluídas (cujos resultados já foram recebidos) e o relançamento de tarefas em andamento. Com as informações referentes à execução de tarefas já escalonadas, o procedimento de recuperação pode tentar a obtenção dos resultados destas, refazendo

conexões com as máquinas. O insucesso desse procedimento resulta no re-escalonamento destas, em máquinas disponíveis. Assim, a perda máxima pós-falha corresponderia a reiniciar todas as tarefas que estavam em andamento e realizar as demais ainda não escalonadas. O melhor cenário de recuperação pós-falha é aquele que permite utilizar todos os resultados já obtidos das tarefas prontas e retomar o andamento daquelas tarefas que estavam em execução.

O *checkpointing*, ou seja, o salvamento dos *checkpoints* é definido por dois tipos de condições (eventos), os quais correspondem às mudanças de estados mais significativas durante a operação de escalonamento. A primeira condição está relacionada ao escalonamento de uma nova tarefa: corresponde ao início da execução, após a tarefa ser transferida para uma máquina do *grid* com sucesso. Seu objetivo é o de manter atualizado o mapa das tarefas distribuídas em máquinas do *grid*. A segunda condição ocorre quando a *home machine* recebe o resultado de uma tarefa, ao final de sua execução. Este resultado pode ser: a) correto e com a resposta produzida; ou b) um indicativo de erro.

Na primeira condição, as mudanças afetam somente a estrutura de dados. Mas, na segunda, a tarefa que completa com sucesso retorna os resultados produzidos, através de transferência de dados para a *home machine*. Estes resultados devem ser armazenados como parte do *checkpoint* para poderem ser reutilizados caso haja falha na *home machine*. A estrutura que armazena as informações referentes ao *checkpoint* possui tamanho variável, dependente do número de tarefas do *job*, da quantidade de tarefas concluídas e dos arquivos de saída. Em conseqüência a esse tamanho, a largura de banda e a latência da rede vão influenciar no tempo de transferência do *checkpoint*, além das variáveis de tempo para salvar tais estruturas em armazenamento estável ou em uma máquina remota. Em caso de receber a indicação de erro, a tarefa deverá ser removida da tabela de tarefas em execução e re-escalonada.

3. SimGrid

A ferramenta desenvolvida para auxiliar na simulação de *grids*, SimGrid [1][2][14], oferece controle de baixo nível sobre os processos e já foi utilizada pelos desenvolvedores do MyGrid para avaliação do comportamento de diferentes estratégias propostas para o escalonador [12], razão pela qual foi também escolhida para este trabalho. SimGrid é um pacote escrito em linguagem C que provê funções básicas e abstrações para a simulação de aplicações em ambientes distribuídos heterogêneos, oferecendo suporte ao estudo de algoritmos de escalonamento e políticas de gerência de recursos em *grids* computacionais.

SimGrid permite a manipulação de tarefas e recursos. Na configuração, as tarefas dividem-se em tarefas de computação ou de transferência de dados, as quais são escalonadas nos recursos de processamento (CPU) ou canal de comunicação, respectivamente. Os recursos CPU e canal de comunicação são tratados como recursos não relacionados; esta associação fica sob responsabilidade do usuário, para atender aos requisitos específicos da simulação em estudo. Assegurar a correspondência adequada entre tarefas de computação / processadores e tarefas de transferência de dados / canais de comunicação para efeito de escalonamento também ficam a cargo do usuário.

Os recursos são modelados com duas características de desempenho: latência (tempo em segundos para acessar o recurso) e taxa de serviço (número de unidades de trabalho executadas por unidade de tempo).

Duas estruturas de dados são oferecidas pelo SimGrid: `SG_Resource()` para gerenciar recursos e `SG_Task()` para gerenciar as tarefas. Um recurso é descrito por um nome, um conjunto de métricas de desempenho e *traces* ou valores constantes de configuração. Por exemplo, um processador é descrito pela medida de sua velocidade (relativa a um processador de referência) e um *trace* sobre a sua disponibilidade, isto é, a porcentagem de CPU que será alocada para um novo processo por um determinado tempo. Um canal de comunicação também é descrito por um nome, um custo e um estado. No caso de tarefas de transferência de dados, o custo é definido pelo tamanho dos dados (em bytes); tarefas de computação têm seu custo expresso pelo tempo (em segundos) de processamento exigido no processador de referência. Finalmente, o estado é usado para descrever o ciclo de vida das tarefas (não escalonada, escalonada, pronta, rodando e completa) [2].

Além disso, a API (*Application Programming Interface*) do SimGrid provê funções básicas para operar os recursos e as tarefas (criação, destruição, inspeção, etc.), funções para descrever as possíveis dependências entre tarefas, e funções para escalonar e remover (desescalonar) tarefas escalonadas previamente nos recursos.

A função `SG_Simulate()` é usada para executar a simulação, conduzir as tarefas através de seus ciclos de vida e simular o uso dos recursos. A simulação pode ser executada durante uma quantidade (virtual) pré-estabelecida de segundos, ou até que uma dada tarefa, qualquer uma, ou todas as tarefas sejam completadas. Em todos os casos, o `SG_Simulate()` retorna uma lista das tarefas que tenham sido completadas desde o último tempo em que foram escalonadas. A função `SG_getClock()` retorna o tempo global virtual da simulação; pode ser chamada a qualquer tempo durante a simulação [2].

A atual API do SimGrid (v2) possui duas interfaces: SG e MSG. A primeira (SG), usada neste trabalho, é a API original; é a de mais baixo nível, sendo bastante flexível e geral. Nesta, a simulação explicita o escalonamento de tarefas em recursos. Já a MSG é construída sobre a SG. Esta camada implementa simulações realísticas baseadas na SG e é mais orientada à aplicação, onde a simulação é construída em termos de comunicação de agentes [1].

4. Modelo de Simulação

O modelo de simulação é composto por três partes: o *job* (aplicação), o ambiente (*grid*) e o escalonador de tarefas.

O *job* é uma lista de tarefas que compõem a aplicação. Cada tarefa possui um custo computacional, o qual corresponde ao tempo necessário para ser executada no processador de referência dedicado. Um processador pode executar (processar) tarefas em diferentes velocidades, mas um processador de referência tem velocidade igual a 1. Cada tarefa tem seu tamanho associado, que representa os dados a serem transferidos pelo canal de comunicação para o recurso onde ela foi escalonada e será executada.

O ambiente (*grid*) é um grupo de sites (*hosts*), ou conjunto de máquinas, compostos por um ou mais processadores, cada um com sua velocidade de operação, que podem a executar tarefas de computação. Além disso, cada processador tem como configuração a sua disponibilidade, cujo valor pode variar entre 0 e 1.

O escalonador é o processo responsável pela distribuição das tarefas através dos processadores disponíveis no *grid*. A estratégia de escalonamento executada pelo escalonador simulado é a WQ (*Workqueue*), descrita na seção 2.1.

Uma característica importante na criação de simulação é a possibilidade de associar dependências entre tarefas. No SimGrid, a associação entre duas tarefas é feita através da função `SG_addDependency()`.

Na simulação desenvolvida, para cada tarefa computacional (*computation task*) é também criada implicitamente uma tarefa de transferência (*transfer task*) baseada no tamanho do arquivo a ser transferido. Esta tarefa de transferência é escalonada no recurso canal de comunicação, antes de iniciar sua tarefa de computação correspondente. Assim, uma tarefa de computação é dependente de uma tarefa de transferência: a tarefa de computação somente irá iniciar sua execução após a tarefa de transferência correspondente ter sido completada. Tais abstrações de dependência visam representar mais adequadamente a execução de uma aplicação em um ambiente real.

4.1. Estudo de caso

O principal objetivo das simulações foi avaliar o impacto no desempenho (tempo total de execução) ocasionado pelas ações referentes ao *checkpointing* no MyGrid. Assim, foram criados dois tipos de aplicações, uma composta por um conjunto de tarefas com custos heterogêneos; e a outra com custos homogêneos. O detalhamento destas será apresentado na seção 5.

Como as simulações foram feitas usando a estratégia de escalonamento *Workqueue*, as tarefas são escalonadas nos primeiros processadores disponíveis. No modelo adotado, o salvamento de estado é efetuado na fase de escalonamento de cada tarefa e ao término destas. Assim, para representar o custo associado aos *checkpoints*, no início da simulação são implicitamente criadas duas tarefas (representando as diferentes fases) que são associadas a cada tarefa da aplicação. Nas tarefas inseridas associadas ao final da execução da tarefa da aplicação, além do custo normal referente às estruturas do *checkpoint*, é associado também o custo relativo a transferência e salvamento dos resultados calculados ou produzidos pela tarefa da aplicação. Essa nova tarefa possui estrutura e funcionamento análogos a uma tarefa normal da aplicação.

O salvamento de estados pode ser feito na máquina local (*home machine*) ou em uma máquina remota, e estas duas opções foram consideradas na simulação. Na prática, o primeiro caso é eficaz apenas para falhas temporárias, enquanto o segundo é adequado para falhas que afetem a *home machine* de forma permanente. No caso de simulação com salvamento na *home machine*, ou seja, na mesma máquina onde o escalonador está sendo executado, a tarefa que representa o *checkpoint* é representada por uma *computation task*, adicionada de um custo computacional que representa o salvamento de estados. Não há necessidade de explicitar o tamanho e o custo de transferência porque a tarefa é executada na máquina local. Assim, todos os recursos podem ser acessados por um único canal de comunicação compartilhado.

Nesse estudo, a *home machine* não executa tarefas da aplicação, mas apenas de escalonamento. Com a introdução do *checkpointing*, a *home machine* fica também encarregada do salvamento destas informações.

Para o caso de simulação da perda de dados e retomada em caso de colapso da *home machine*, os resultados das tarefas devem ser salvos em armazenamento estável, aqui representado por uma máquina remota (já que não é esta que foi atingida pelo defeito). Neste caso, o tempo gasto para a transferência também é computado nas simulações. Dessa forma a tarefa que representa o *checkpoint* é composta por duas partes – uma *transfer task* e uma *computation task* – como uma tarefa normal da aplicação. Nesse caso, a *home machine* executa as funções do escalonador e transfere as estruturas de

checkpoint e as respostas das tarefas para serem salvas na máquina remota. Assim ocorre o compartilhamento do canal de comunicação entre a máquina remota e a *home machine*; logo, essa disputa pelo recurso é refletida no resultado final da simulação. A máquina remota criada no início da simulação é exclusiva para a execução de tarefas que representam o *checkpoint*: ela não pertence ao conjunto de máquinas que executam tarefas da aplicação, como pode ser visto no exemplo de execução mostrado na Figura 1 a). Na prática, esta máquina pode abrigar um sistema de armazenamento robusto.

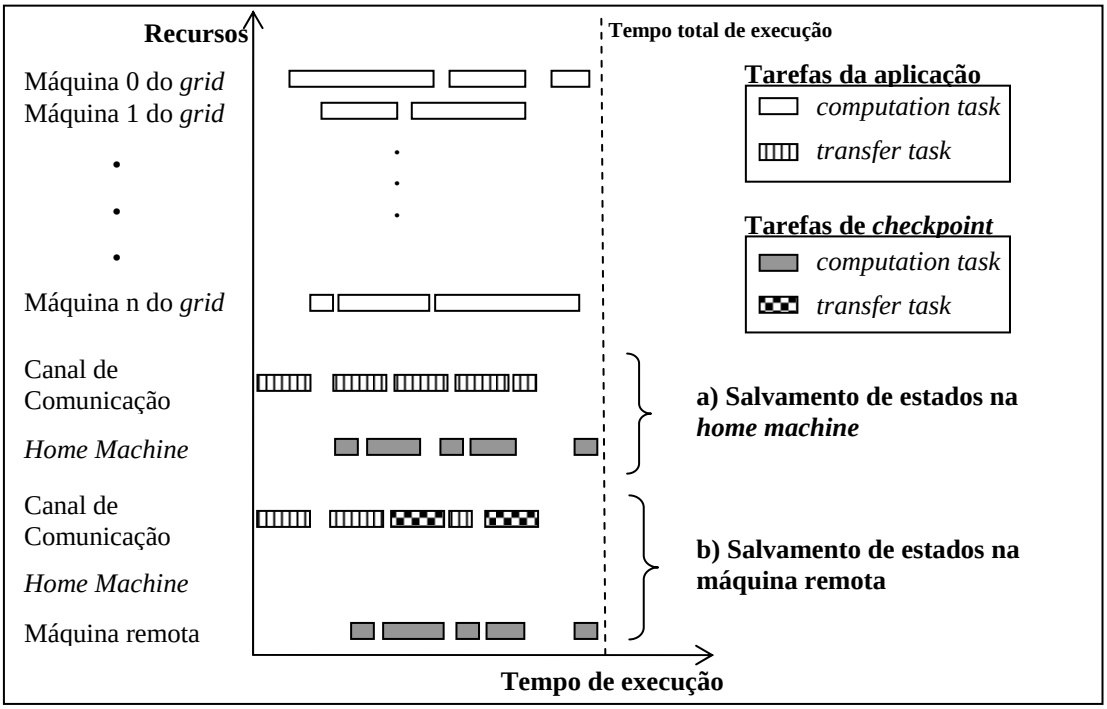


Figura 1. Execução de uma aplicação com *checkpoint*

A saída da simulação corresponde ao tempo total de execução da aplicação. Nos experimentos efetuados com a execução do *checkpointing*, este tempo reflete a sobrecarga devido aos *checkpoints*, porém mostra que os totais dos tempos de execução do *checkpointing* não se somam simplesmente ao tempo de execução das aplicações isoladas (sem *checkpointing*). Isso ocorre porque as tarefas de transferência e computação dos *checkpoints* – expressas através do canal de comunicação e dos ciclos de CPU – são executadas de forma concorrente com outras tarefas da aplicação como pode ser observado na Figura 1 b), mas seguem a ordem de execução segundo suas dependências. O uso do simulador auxilia na percepção deste comportamento referente aos recursos consumidos.

O impacto dos procedimentos de *checkpointing* sobre o escalonador é mais significativo do que sobre as aplicações, já que ele está envolvido em toda coordenação do processo e no controle da execução de suas ações. Entretanto, ao usuário do *grid* vai transparecer o tempo de obtenção dos resultados de suas aplicações. Por isso, este foi o parâmetro de observação escolhido.

5. Resultados Experimentais

O ambiente de simulação criado para executar os experimentos é formado por uma *home machine* e nove máquinas monoprocessadas das quais uma, particularizada como

máquina remota, armazena o *checkpoint*. Todas têm capacidade de processamento igual a 1. O *job* da aplicação é composto por um conjunto de tarefas com tamanho fixo em 100K *bytes*, mas cuja quantidade varia exponencialmente de 4 até 512.

Cada tarefa da aplicação possui, além de sua configuração, o custo computacional (expresso em segundos) e o tamanho (em *bytes*) associados, relativos às estruturas dos *checkpoints* que devem ser transferidas e salvas. Tais valores, utilizados para configurar o simulador, foram estabelecidos a partir de medidas efetuadas sobre uma implementação preliminar do mecanismo de *checkpointing* no MyGrid; eles podem ser observados na Tabela 1. O método usado baseou-se na criação de *jobs* com diferentes quantidades de tarefas, que eram executados no MyGrid. O *job* foi composto por tarefas que calculam a seqüência de Fibonacci até o número 35. Além dos tempos (custo do *checkpoint*), foram observados também as características das estruturas de dados relacionadas. Particularidades das tarefas que compõem o *job* (tamanho e complexidade) não influenciam no dimensionamento da estrutura do *checkpoint*; ele armazena informações de interesse do escalonamento e das respostas recebidas. Estas últimas dependem das características das aplicações.

Tabela 1. Parâmetros de *checkpoint* medidos em execuções reais

Nº de Tarefas	Tamanho (bytes)	Tempo de transferência (ms)
4	2249	81
8	3593	99
16	6287	105
32	11663	118
64	22409	184
128	43919	284
256	86927	445
512	172943	771

Com objetivo de simplificar os experimentos, a simulação idealiza as máquinas e a rede de comunicação (usada para transferência dos dados): todas as máquinas têm disponibilidade de 100%, a rede tem largura de banda de 10Mbits/s e latência zero.

O tempo total de execução da aplicação, definido como sendo o intervalo de tempo entre o momento em que a primeira tarefa do *job* é iniciada e o primeiro instante em que todas as tarefas finalizaram suas execuções, foi a métrica de desempenho adotada neste trabalho. Dentro deste tempo também é contabilizado o tempo gasto na transferência de tarefas e estruturas do *checkpointing*, quando este é efetuado.

Uma intuição que embasou os experimentos é a de que os procedimentos de *checkpoint* teriam maior impacto sobre um conjunto de tarefas homogêneas do que sobre as heterogêneas. A quase sincronização do término de muitas tarefas causaria uma rajada de *checkpoints*. Assim, foram feitas simulações comparativas com estes dois cenários.

5.1. Tarefas Heterogêneas

Neste experimento, foram executados 8 *jobs*, cada um composto por 4, 8, 16, 32, 64, 128, 246 e 512 tarefas respectivamente, onde o custo de cada tarefa dentro de cada *job* segue uma distribuição pseudo-aleatória uniforme no intervalo de 60 a 3600 segundos. Atendendo a essas configurações, dois cenários foram simulados: num cada tarefa produzia arquivos de saída com tamanho de 500Kb e, noutro, este tamanho era de 100Kb. Esses dois cenários foram comparados com um terceiro, onde a execução das

tarefas era feita sem a geração de *checkpoints*. Os tempos de transferência e tamanhos de estruturas de *checkpoint* efetuados na fase de escalonamento de uma determinada tarefa seguem a Tabela 1, enquanto que os *checkpoints* efetuados ao final da execução desta tarefa têm acrescidos aos valores da Tabela 1 o tamanho da resposta e do tempo para transferência desta. Esses tempos também foram obtidos pelas médias dos tempos de transferências reais na rede de 10Mbits/s: para 100Kb obteve-se o tempo de 314 ms e para 500Kb, 1317 ms. Vale ressaltar que estes valores variam em função do tráfego da rede. Os tempos totais de simulação para cada *job*, nas diferentes configurações, estão representados no gráfico da Figura 2.

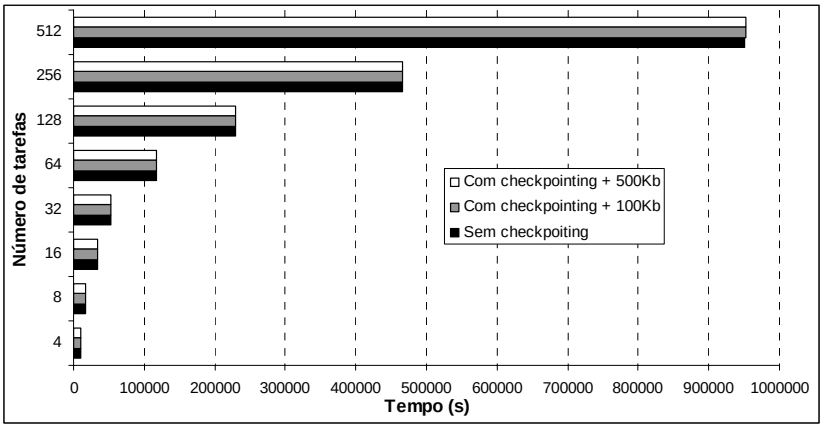


Figura 2. Gráfico com os tempos das tarefas heterogêneas (60-3600s)

Pode-se notar que a degradação de desempenho ou o acréscimo no tempo total da simulação, causado pela inclusão do mecanismo de *checkpointing*, foi baixo, variando de 0,02 a 0,12% para tarefas que produzem arquivos de 100Kb, e 0,06 a 0,19% para tarefas que produzem arquivos de 500Kb.

5.2. Tarefas Homogêneas

O segundo tipo de experimento segue as mesmas configurações do primeiro, porém todas as tarefas são homogêneas e possuem custo fixo de 120 segundos. Esses experimentos não estão associados necessariamente a cenários em que as tarefas são idênticas e as máquinas homogêneas, mas podem estar associadas a cenários heterogêneos nos quais os resultados sejam recebidos em rajadas (muitas tarefas retornam resultados quase simultaneamente). Os tempos totais de simulação em segundos para cada *job*, nas diferentes configurações, podem ser observados no gráfico da Figura 3.

Os resultados, análogos aos conjuntos de tarefas heterogêneas, apresentam baixa degradação de desempenho quando do uso de *checkpointing*, variando entre os percentuais de 0,46 a 1,83 para tarefas que produzem arquivos de 100Kb, e de 1,56 a 2,92% para tarefas que produzem arquivos de 500Kb.

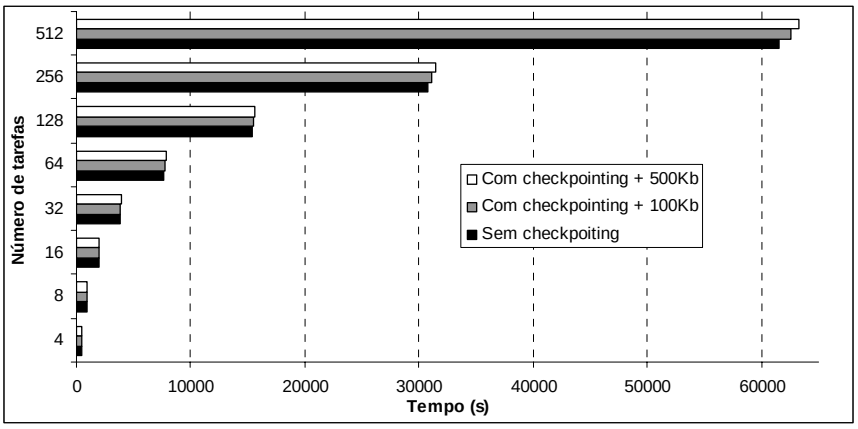


Figura 3. Gráfico com os tempos das tarefas homogêneas (120s)

6. Conclusão

Neste trabalho, considera-se a probabilidade real de falhas em ambientes de *grids*, e propõe-se o uso de *checkpoints* para aumentar a tolerância a falhas do sistema. Estes *checkpoints*, vinculados ao escalonador do *grid*, registram informações referentes à distribuição, localização e resultados das tarefas em execução. Em caso de falha, o escalonador pode ser reativado, na máquina original ou em outra máquina do *grid*, reduzindo significativamente a perda de computação já realizada.

O estudo aqui realizado visou avaliar o desempenho da técnica em diferentes configurações de aplicações possíveis no *grid*. O ambiente que serviu para a modelagem foi o MyGrid e a ferramenta empregada para as simulações foi o SimGrid.

O uso de ferramentas de simulação, tal como o SimGrid, é uma boa alternativa para compreender o comportamento de escalonadores de tarefas, além de ser útil para análise do impacto de técnicas como o *checkpointing*. Mostrou-se bastante flexível, mas apresentou um certo grau de complexidade na implementação das simulações.

Foram simulados cenários com topologia fixa e variações no número e custo das tarefas. Os testes revelam o custo adicionado ao tempo original das aplicações, tanto para o conjunto de tarefas homogêneas, quanto heterogêneas. Pode-se notar que os resultados causados pela inclusão do *checkpointing* no escalonador na execução de conjuntos de tarefas heterogêneas e homogêneas mostraram-se semelhantes, causando baixa degradação no desempenho. O percentual máximo de 2,92% de aumento no tempo total de execução de um conjunto de tarefas (BoT) é aceitável no uso da técnica.

Além disso, vale lembrar: a) que não há necessidade do usuário introduzir qualquer tipo de alteração na programação das aplicações – portanto a técnica fica transparente aos usuários do *grid*; b) o tempo de recuperação pós-falhas deverá ser bastante reduzido, pois se restringe a reiniciar o escalonador, carregando as estruturas de dados anteriormente salvas. Resultados mais completos poderão ser obtidos observado-se outras características da aplicação, tais como: a granulosidade das tarefas, sua quantidade, o custo e o tamanho das respostas produzidas.

Agradecimentos

Este trabalho é parcialmente financiado pela HP Brasil P&D, a quem os autores agradecem.

Referências

[1] Casanova, H.; Legrand, A.; Marchal, L. **Scheduling Distributed Applications: the SimGrid Simulation Framework**. Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid'03), 2003.

[2] Casanova, H. **Simgrid: A Toolkit for the Simulation of Application Scheduling**. Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid'01), p. 430-437, May 2001.

[3] Cirne, W.; Paranhos, D.; Costa, L.; Santos-Neto, E.; Brasileiro, F.; Sauvé, J.; Silva, F. A. B.; Barros, C. O.; Silveira, C. **Running Bag-of-Tasks Applications on Computational Grids: The MyGrid Approach**. Proceedings of the ICCP'2003 - International Conference on Parallel Processing - October 2003.

[4] Cirne, W.; Brasileiro, F.; Sauvé, J.; Andrade, N.; Paranhos, D.; Santos-Neto, E.; Medeiros, R. **Grid computing for bag of tasks applications**. Proceedings of 3rd IFIP Conference on E-Commerce, E-Business and E-Government - Sept. 2003.

[5] Cirne, W. **Grids Computacionais: Arquiteturas, Tecnologias e Aplicações**. III ERAD - Escola Regional de Alto Desempenho, Santa Maria, RS. Janeiro 2003.

[6] Foster, I. **What is the Grid? A Three Point Checklist**. GRID today online magazine, July 2002.

[7] Foster, I.; Kesselman, C.; Tuecke, S. **The anatomy of the Grid: Enabling scalable virtual organizations**. LNCS, 2150:1. 2001.

[8] Medeiros, R.; Cirne, W.; Brasileiro, F.; Sauvé, J. **Faults in Grids: Why are they so bad and What can be done about it?**. Grid Computing, 2003. Proceedings. Fourth International Workshop, p. 18-24, Nov. 17, 2003.

[9] MyGrid Homepage: **MyGrid Online Manual**. In: http://www.ourgrid.org/mygrid.

[10] Paranhos, D.; Cirne, W.; Brasileiro, F. **Trading Cycles Information: Using Replication to Schedule Bag-of-Tasks Applications on Computational Grids**. Proceedings of the Euro-Par 2003: International Conference on Parallel and Distributed Computing. August, 2003.

[11] Pradhan, D. K. Fault-Tolerant Computer System Design. Upper Saddle River: Prentice Hall, 1996.

[12] Santos-Neto, E. **Escalonamento de Aplicações que Processam Grandes Quantidades de Dados em Grids Computacionais**. 2004. 71 f. Dissertação (Mestrado em Informática) – CCT – UFCG, Campina Grande.

[13] Santos-Neto, E.; Cirne, W.; Brasilero, F.; Lima, A. **Exploiting Replication and Data Reuse to Efficiently Schedule Data-Intensive Applications on Grids**. Proceedings of the 10th Workshop on Job Scheduling Strategies for Parallel Processing, June 2004.

[14] **SimGrid Homepage**. In: http://gcl.ucsd.edu/simgrid.

Um Injetor de Falhas de Comunicação construído usando Programação Orientada a Aspectos*

Karina Kohl Silveira, Taisy Silva Weber

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

{kohl,taisy}@inf.ufrgs.br

***Resumo**: Aplicações de alta dependabilidade devem estar aptas a detectar falhas e corrigir erros para que sejam evitados danos irreparáveis a vida, ao meio ambiente ou aos negócios. Uma técnica usada para garantir operação adequada de mecanismos tolerantes a falhas é a injeção de falhas. FICTA é uma ferramenta de injeção de falhas de comunicação para a validação de aplicações distribuídas baseadas em UDP e desenvolvidas em Java. FICTA utiliza Programação Orientada a Aspectos, a qual permite que comportamentos afetando diversas classes do sistema, os chamados interesses cruzados, sejam reunidos em um único módulo chamado aspecto. A abordagem baseada em AOP permitiu a construção de uma ferramenta altamente modular, reusável e que não causa intrusividade espacial nas aplicações alvo.*

***Abstract**: High dependable applications detect faults and correct errors to avoid risks to life, environment and business that depend on them. Fault injection is an efficient technique to test fault tolerance mechanisms. We develop FICTA, a fault injector to validate UDP-based distributed network applications programmed in Java. FICTA uses aspect oriented programming, that permits behavior that affects a variety of classes to be joined in a single module called aspect. The AOP-based approach allows the implementation of a high modular and reusable tool that is not intrusive in target applications.*

1 Introdução

Aplicações distribuídas estão sujeitas a falhas que afetam o sistema de troca de mensagens na rede de comunicação. Para alcançar dependabilidade, é necessária a implementação de mecanismos de detecção e remoção das falhas que afetam mensagens ou nós do sistema. Além disso, é necessário que esses mecanismos sejam validados para assegurar que satisfazem sua especificação. A ausência de testes desses mecanismos pode levar a comportamentos inesperados e errôneos na fase operacional da aplicação. Uma etapa fundamental no desenvolvimento destes sistemas é a fase de validação sob falhas, na qual é observado o comportamento da aplicação em vários cenários de falhas emuladas.

A injeção de falhas pode ser definida como a introdução intencional e controlada de falhas em uma aplicação alvo para observar seu comportamento [ARL90]. As falhas injetadas permitem avaliar se a aplicação sob teste se comporta conforme especificado na presença de falhas, além de determinar se a cobertura de falhas dos mecanismos

* Financiado pelos projetos ACERTE/CNPq (472084/2003-8) e DepGriFE/HP Brasil P&D

implementados está adequada para a confiabilidade e a disponibilidade exigidas. Para esse trabalho interessa a injeção de falhas implementada por software, onde falhas são injetadas por software corrompendo código ou dados ou alterando o fluxo de operações.

A crescente necessidade de tornar aplicações distribuídas cada vez mais confiáveis exige que seja ampliado o uso da técnica de injeção de falhas. Para isso é necessário o estudo de abordagens que levem a soluções modulares, reusáveis e que causem o mínimo de intrusividade espacial nas aplicações alvo, visto que essas características facilitam e ampliam a utilização de injeção de falhas.

Baseado nessas necessidades, esse artigo apresenta uma nova abordagem, baseada em Programação Orientada a Aspectos [KIC97] para a validação de aplicações distribuídas escritas em Java baseadas no protocolo UDP. A abordagem se baseia na instrumentação de métodos de envio e recepção de mensagens dos protocolos usados pela aplicação, simulando falhas de comunicação. O artigo apresenta conceitos de injeção de falhas por instrumentação de código e de orientação a aspectos, discute como a orientação a aspectos pode auxiliar no desenvolvimento de injetores de falhas, apresenta o protótipo de uma ferramenta construída com essa abordagem e demonstra sua utilização em um caso exemplo. Além disso, a abordagem e a ferramenta apresentadas nesse trabalho se preocupam em evitar a intrusividade espacial, comum em algumas abordagens de injeção de falhas baseadas em *software*. Finalmente, são tecidas algumas considerações sobre os resultados alcançados.

2 Redução da intrusividade espacial através de instrumentação de código

A interferência do injetor de falhas da aplicação alvo, chamada de intrusividade, pode ser observada de forma temporal e espacial [DAW96]. Na intrusividade temporal, o tempo de execução da aplicação é aumentado devido ao acréscimo das atividades do injetor, que são executadas juntamente com a aplicação alvo. A espacial surge da modificação do código da aplicação alvo. A intrusividade temporal é crítica principalmente em aplicações de tempo real, porém a intrusividade espacial é uma preocupação para todas as classes de aplicação, pois alterações no código fonte podem espalhar *bugs* e mesmo alterar o fluxo de execução da aplicação. Adicionalmente, não é raro que a totalidade ou uma parte considerável do código fonte não esteja disponível para teste.

A instrumentação de código permite que módulos de programas sejam completamente reescritos em tempo de execução facilitando a introdução de novas funcionalidades. É possível a introdução ou remoção de instruções, alterando o uso de classes, variáveis e constantes. Esse trabalho usa instrumentação de código para a inserção de falhas nas aplicações alvo. Várias técnicas vêm sendo estudadas para evitar a alteração do código original da aplicação, como por exemplo, reflexão computacional e programação orientada aspectos.

A reflexão computacional oferece flexibilidade à instrumentação de código, pois permite alteração nas classes de uma aplicação independente da disponibilidade do código fonte. Porém, dependendo da ferramenta de reflexão utilizada, uma cláusula de instanciação deve ser declarada ao longo da declaração da classe base (por exemplo, a ferramenta OpenJava), introduzindo sobrecarga e custos em manutenção e alteração [ROY2003].

A instrumentação por AOP permite a interceptação e instrumentação de elementos de classes, necessitando apenas o conhecimento da API das aplicações. Além disso, preserva a

integridade da classe base, isto é, não são necessárias mudanças estruturais na base. A AOP possui elementos que identificam claramente os nomes de cada método em cada classe que está envolvida no processo de cruzamento, dessa forma o programador pode identificar facilmente a parte do código que necessita ser atualizada após a chamada de cada método [ROY2003]. A principal vantagem do uso de AOP em relação à reflexão computacional é que sua funcionalidade é plugável. Se uma funcionalidade necessita ser removida do sistema, basta que se remova o aspecto responsável pela funcionalidade. Ao contrário da ferramenta de reflexão computacional OpenJava, que cria a necessidade de varrer cada classe afetada e remover as cláusulas de instanciação

3 Injeção de falhas de comunicação

Uma ferramenta de injeção de falhas por *software* geralmente é um trecho de código que usa todos os ganchos possíveis do processador para criar um comportamento errôneo de maneira controlada [CAR98]. Para esse trabalho, interessam as falhas de comunicação que podem ocorrer em sistemas distribuídos suportados por uma infra-estrutura de rede. Estas afetam as mensagens que são transmitidas por um canal de comunicação, que podem ser omitidas, duplicadas ou entregues com atraso.

Diversas ferramentas de injeção de falhas têm sido desenvolvidas [HSU97]. A maior parte destas ferramentas é específica para um determinado sistema alvo, não permitindo reuso. Várias são experimentos acadêmicos e não estão disponíveis para avaliação ou aplicação. Poucas dessas ferramentas ([DAW96], [JAC2004a]) são injetores de falhas de comunicação. Jaca e FIONA são ferramentas bastante próximas a FICTA. Jaca [MAR2002] é uma ferramenta extensível com a finalidade de validar aplicações em Java. Jaca é baseada em reflexão computacional e sua arquitetura segue um padrão de software projetado para ferramentas de injeção de falhas. Como plataforma para reflexão computacional, Jaca utilizada o protocolo de meta objetos não padrão Javassist, que permite que os *bytecodes* sejam transformados durante o tempo de carga. Jaca teve seu modelo de falhas estendido para injetar falhas de comunicação UDP [JAC2004b]. Essa extensão necessita uma fase de pré-processamento do código da aplicação (código fonte ou *bytecode*) para a injeção de falhas. Isto é necessário para que a aplicação possa se referir às classes de comunicação que suportam a injeção de falhas. FIONA [JAC2004a] tem um comportamento semelhante à extensão de Jaca, mas utiliza JVMTI (*Java Virtual Machine Tool Interface*). JVMTI é uma nova interface de programação nativa da plataforma Java para o desenvolvimento de ferramentas de monitoramento e depuração, permitindo a instrumentação de aplicações Java. O uso de JVMTI faz que não seja necessária a alteração da máquina virtual nem da aplicação, pois é uma biblioteca dinâmica que é carregada opcionalmente pela máquina virtual quando é inicializada.

A abordagem desse trabalho diferencia-se de FIONA, Jaca e também das demais ferramentas de injeção de falhas de comunicação conhecidas por usar AOP para realizar a interceptação e instrumentação de *bytecodes* das classes que implementam os métodos de comunicação das aplicações alvo distribuídas. AOP é um paradigma de programação relativamente novo, complementar à orientação a objetos e que tem como objetivo a melhor modularização de interesses que cruzam várias classes de um sistema. A utilização de abordagem AOP permite que os aspectos sejam carregados junto com a aplicação alvo (diz-se que os aspectos e as classes são integrados em tempo de carga) e a instrumentação das primitivas de comunicação é realizada efetivamente em tempo de execução.

4 Programação orientada a aspectos

AOP separa interesses que afetam diversos módulos de um sistema, em unidades únicas chamadas aspectos. Esses ”interesses cruzados” são chamados *crosscutting concerns*. Um *concern* é um comportamento específico, um conceito ou uma área de interesse. Como exemplo pode ser citada a manipulação de mecanismos de *logging*, autenticação, segurança, persistência, etc. Os *crosscutting concerns* tendem a afetar vários módulos de implementação de um sistema, isto é, seus códigos se espalham em vários módulos, dificultando a sua manutenção e compreensão [LAD2002].

Normalmente, as diversas implementações para AOP utilizam construções chamadas *advices* para descrever o código que deve ser inserido em localizações chamadas de *join points*. O código pode ser inserido antes ou depois dos *join points*. Além disso, um *around advice* permite que o desenvolvedor especifique código que deve substituir o código original que é executado naquele *join point*. Também existem as construções chamadas de *pointcuts,* que especificam em quais *join points* o código deve ser executado. Um aspecto, por definição, é uma unidade modular, que encapsula os conceitos anteriores e que corta a estrutura de outras unidades. Um aspecto é similar a uma classe tendo um tipo, estendendo outras classes e outros aspectos. Ele pode ser abstrato ou concreto e ter campos, métodos e tipos como membros. A AOP permite a implementação individual de comportamentos, de forma fracamente acoplada, e a combinação dessas implementações para formar o sistema final. De fato, cria um sistema usando implementações modulares, fracamente acopladas, de comportamentos cruzados.

4.1 Orientação a aspectos versus orientação a objetos

A unidade modular de AOP é o aspecto [LAD2002]. Na Programação Orientada a Objetos, OOP, a unidade natural modular é a classe, e um comportamento cruzado está espalhado em várias classes. Trabalhar com código que aponta para responsabilidades que atravessam o sistema gera problemas que resultam na falta de modularidade. A AOP complementa a OOP por facilitar um outro tipo de modularidade que expande a implementação espalhada de um comportamento dentro de uma simples unidade. Como resultado, um único aspecto pode contribuir na implementação de procedimentos, módulos ou objetos, aumentando a reusabilidade dos códigos.

O processo de criação e integração de aspectos com as classes que devem ser afetadas por esses aspectos é chamado de *weaving*. O *weaving* realiza o processo de instrumentação das classes, modificando o *bytecode* de modo a incluir interfaces e chamadas aos *advice*s. Os integradores de aspectos (*aspect weavers*) devem processar as classes componentes dos sistemas com os aspectos, compondo-os de forma apropriada para produzir a operação total desejada do sistema.

O processo de *weaving* pode ser estático ou dinâmico. Com o *weaving* estático, os aspectos são integrados resultando em uma única classe, os aspectos são entidades em tempo de compilação. Em *weaving* dinâmico, os aspectos sempre são classes separadas, que são chamadas por algum *framework* que intercepta dinamicamente o uso de algum objeto, como é realizado, por exemplo, pelo AspectWerkz [BON2004]. Nesse caso, os aspectos são entidades em tempo de execução.

4.2 Um framework para AOP

AspectWerkz é um *framework* dinâmico de programação orientada a aspectos, para Java, e foi usado como base na implementação de FICTA. Qualquer classe Java pode ser um aspecto, ou seja, pode ser definida para ter um comportamento que corta outras classes, significando que ela se torna uma unidade modular para *crosscutting concerns*. Não há necessidade de estender nenhuma classe em especial ou implementar interfaces específicas. Os aspectos podem ser definidos utilizando XML ou anotações. Para esse trabalho importam aspectos definidos em arquivos XML, pois se tornam mais apropriados para construir aspectos para injeção de falhas sem intrusividade e de fácil configuração, adição e remoção.

AspectWerkz trabalha com *weaving* dinâmico e para isso oferece mecanismos chamados de modo *offline* e modo *online*. No modo *offline* as classes sofrem uma pós-compilação antes da sua execução para que os aspectos sejam integrados nas aplicações, nesse caso são necessárias duas fases para gerar as classes. A primeira fase é uma compilação padrão utilizando o compilador padrão Java (`javac`). Na segunda fase é executado o compilador do AspectWerkz no modo *offline,* apontando para as novas classes criadas. O compilador modifica os *bytecodes* das classes incluindo os *advices* nos *join point* corretos. Esse processo assemelha-se aos mecanismos de *weaving* estático, necessitando que todas as classes sejam recompiladas para que os aspectos sejam efetivamente integrados as classes.

No modo *online*, as classes são modificadas durante sua execução de forma transparente. Para o modo *online*, AspectWerkz está preso ao mecanismo de baixo nível de carregamento de classes da JVM, permitindo interceptar todas chamadas de carregamento de classes e transformando os *bytecodes* em tempo de execução. Para isso, AspectWerkz modifica o mecanismo de carga de classes da JVM utilizando *HotSwap*, no caso de Java 1.4, ou JVMTI no caso de Java 1.5. De forma simplificada, tanto *HotSwap*, quanto JVMTI encapsulam a habilidade de substituir o código instrumentado em uma aplicação que está executando, através do mecanismo de depuração ou de processos chamados de agentes.

O *framework* AspectWerkz foi escolhido pelo seu mecanismo de *weaving* dinâmico *online*. Para uma abordagem de injeção de falhas, que visa evitar intrusividade, essa característica é muito importante. Os sistemas sob teste nem sempre possuem seu código disponível para ser alterado e recompilado. Além disso, a recompilação de todas as classes do sistema alvo pode ser um processo oneroso, ou seja, o *weaving* estático não seria uma boa solução. O *framework* AspectWerkz permite que os aspectos alterem os *bytecodes* das classes alvo em tempo de carga ou de execução, atingindo todas as classes que se deseja instrumentar, sem necessidade de serem compiladas especificamente com a ferramenta do *framework.*

5 Uso de AOP em injeção de falhas

AOP é uma excelente alternativa para impedir a alteração do código fonte das aplicações a serem testadas, evitando assim intrusividade espacial. A AOP oferece um mecanismo para a injeção de falhas de alto nível em sistemas orientados a objetos, possibilitando a instrumentação de métodos, classes e variáveis de um sistema, em tempo de compilação, carga ou execução, sem a necessidade de alterar a estrutura do sistema, tornando indiferente a disponibilidade do código fonte para o teste. Dessa forma, é necessário apenas o mínimo de informação sobre a aplicação, como classes, métodos e nomes de atributos. Com o uso

de AOP a instrumentação é encapsulada em um aspecto e é introduzida em nível de *bytecodes*. O aspecto identifica as chamadas aos métodos que devem ser modificados e instrumenta todo o código ou parte dele em tempo de carga e execução. Além disso, o injetor de falhas pode ser facilmente inserido e removido da aplicação a ser validada, apenas pela remoção dos aspectos relativos à injeção de falhas (simplesmente sem carregar o aspecto no momento que a aplicação é disparada).

5.1 Recursos de AOP para injeção de falhas

A injeção de falhas possui um comportamento que abrange os diversos módulos da aplicação alvo, afetando métodos que são executados em diversas classes em diversos pontos da aplicação. Desta forma, a injeção de falhas pode ser encapsulada sob a forma de aspectos. Os métodos da aplicação que devem sofrer a injeção de falhas quando executados, se encaixam nos conceitos de *join point* e *pointcuts*, os quais sinalizam que aqueles determinados métodos poderão ter comportamento alterado e ou adicionado. Ao serem atingidos esses métodos (*join points*), eles tem seus comportamento alterado pela lógica de injeção de falhas que se encontra nos *advices* do aspecto. A abordagem básica deste trabalho é a implementação de aspectos que definem *pointcuts* em termos das primitivas de comunicação das aplicações e protocolos, e implementam *advices* que substituem os códigos originais por códigos instrumentados.

De acordo com Carreras [CAR98], uma ferramenta de injeção de falhas por software geralmente é um trecho de código que usa todos os ganchos possíveis do processador para criar um comportamento errôneo de maneira controlada. Dessa forma, mostra-se que o comportamento conjunto dos *join points* com os *pointcuts* é compatível com os ganchos definidos por Carreras. Além disso, os *advices* possibilitam a inserção do comportamento errôneo de forma controlada.

Para a injeção de falhas em aplicações distribuídas de rede, é necessário instrumentar as primitivas de comunicação. A AOP é uma alternativa viável e eficiente, pois torna possível a construção de aspectos que interceptam, em um único ponto, os métodos que são chamados de qualquer lugar da aplicação. Além disso, as classes da aplicação não precisam ser alteradas de forma alguma, nem ao mesmo para a inserção de cláusulas de instanciação ou inserção de bibliotecas adicionais. A união das classes da aplicação com os aspectos é realizada pelo *weaver* da ferramenta de programação orientada a aspectos que está sendo utilizada.

5.2 Vantagens da AOP para injeção de falhas

O encapsulamento da lógica de injeção de falhas em um único módulo que afeta as diversas classes do sistema, separando o código funcional do código de injeção, facilita a inserção e remoção dos aspectos na aplicação alvo e auxilia na criação rápida de diversos cenários de falhas. Além disso, caso o sistema deva ser executado sem o injetor de falhas (por já ter sido validado e colocado em produção, por exemplo), basta que se remova o aspecto responsável por essa funcionalidade, conservando intacto o programa original

Essa funcionalidade plugável (facilidade de inserção e remoção) oferecida pela AOP traz outro benefício importante na construção de injetores de falhas. Sistemas de missão crítica, que exigem alta disponibilidade, devem ser fortemente validados antes de serem colocados em produção, pois podem até mesmo envolver risco a pessoas (por exemplo, sistemas de controle aéreo). Porém, além desses, existem sistemas distribuídos sem um alto

grau de risco, mas que nem por isso podem deixar de ser validados. Muitas vezes por falta de tempo no desenvolvimento, ou na falta de ferramentas flexíveis que possam ser facilmente usadas em várias aplicações, a fase de validação dos mecanismos de tolerância a falhas fica relegada a um segundo momento, muitas vezes já na fase de manutenção. Um injetor de falhas, altamente reusável e facilmente (des)plugável é de valor inquestionável, pois amplia o uso da injeção de falhas como mecanismo de validação, fazendo que sistemas sejam mais facilmente validados e ofereçam maior segurança de uso.

6 A ferramenta FICTA

FICTA (*Fault Injection Communication Tool based on Aspects*) tem como principal objetivo a injeção de falhas de comunicação em aplicações Java distribuídas de rede, usando a Programação Orientada a Aspectos. Como resultado de um experimento de teste aplicando FICTA, a aplicação alvo Java, que está sendo validada e que tenha sido construída usando técnicas de tolerância a falhas, terá sua cobertura de falhas determinada para um determinado cenário de teste. Caso a cobertura não seja adequada, o desenvolvedor da aplicação pode refinar os mecanismos construídos. Cenários de falhas podem ser redefinidos e os experimentos podem ser repetidos tantas vezes quanto necessário para o teste dos mecanismos de tolerância a falhas da aplicação alvo.

6.1 Aplicações alvo para o injetor de falhas FICTA

O protocolo UDP (*User Datagram Protocol*) é um protocolo que envia pacotes de dados independentes, chamados datagramas, de um computador a outro, porém sem garantias sobre a chegada desses pacotes. O protocolo UDP é não confiável e não orientado a conexão, no entanto é comumente usado como base para a implementação de protocolos que implementam a confiabilidade necessária através de camadas superiores adicionais. Um exemplo é o *middleware* de comunicação de comunicação de grupo JGroups [BAN98] que usa UDP por padrão na base de sua pilha de protocolos.

FICTA implementa injeção de falhas de comunicação através da instrumentação da classe `java.net.DatagramSocket` da API padrão da linguagem Java. É essa classe que permite o envio e recebimento de pacotes através do protocolo UDP. A instrumentação de uma classe base da linguagem, garante a reusabilidade para toda e qualquer aplicação, protocolo ou *middleware,* escrito em Java e baseada no protocolo UDP. Portanto, qualquer aplicação Java construída sobre essa classe pode ser validada utilizando a ferramenta FICTA. Caso a ferramenta tivesse sido construída para instrumentar uma classe específica da aplicação alvo, não seria genérica o suficiente para garantir reusabilidade. No momento que se instrumenta uma classe da aplicação, restringe-se a ferramenta apenas aquela aplicação. Por aplicação entende-se desde sistemas desenvolvidos diretamente com a API, protocolos, *middlewares* e *toolkits* que se utilizam dessa classe para a construção de outros sistemas.

A classe Java `java.net.DatagramPacket` representa um datagrama UDP. Datagramas são utilizados para a entrega de pacotes em sistemas sem conexão e normalmente incluem o endereço destino e informação de porta. A classe `java.net.DatagramSocket` é um *socket* utilizado para o envio de datagramas em uma rede pelo protocolo UDP. Um datagrama é enviado por um `java.net.DatagramSocket` pela chamada do método `send(DatagramPacket)`. O método `receive(DatagramPacket)` é utilizado para realizar a recepção de um

datagrama. A classe `java.net.MulticastSocket` também pode ser utilizada para envio e recebimento de datagramas para um grupo *multicast*. Essa classe é subclasse de `java.net.DatagramSocket` e adiciona as funcionalidades para *multicasting*. Com FICTA também é possível validar protocolos que utilizam *multicast* UDP, já que a classe `java.net.MulticastSocket` é uma classe filha de `java.net.DatagramSocket` e não reimplementa os métodos *send* e *receive*. Ou seja, a injeção de falhas acontece transparentemente, através do mesmo mecanismo.

6.2 Modelo de falhas e ativação

FICTA considera as falhas de colapso de nós e omissão de mensagens. Omissão de mensagens pode representar tanto falhas no receptor ou emissor, assim como perdas de pacotes na rede de comunicação. Essas falhas não são tratadas pelo protocolo UDP e devem, portanto, ser consideradas no desenvolvimento de protocolos de mais alto nível usadas pela aplicação, ou na própria aplicação alvo.

Uma abordagem comumente usada para a ativação de falhas de comunicação é disparar a falha durante o envio ou recepção de uma mensagem [DAW96]. FICTA utiliza como ativador uma quantidade determinada de mensagens enviadas e recebidas no processo da aplicação distribuída que está sendo executado com o injetor de falhas. Após um determinado número de mensagens enviadas ou recebidas é disparada a falha. Esse número de mensagens é determinado pelo usuário que está realizando o experimento de teste ou pode ser gerado aleatoriamente pelo injetor. O injetor identifica se deve gerar os valores do experimento se receber um parâmetro de entrada chamado "`RANDOM`". Caso o usuário deseje especificar o percentual de mensagens que deve ser utilizado para que as falhas sejam disparadas, o parâmetro "`DETERMINISTIC`" deve ser utilizado. Caso os parâmetros "`RANDOM`" ou "`DETERMINISTIC`" não sejam especificados pelo usuário, é assumido como padrão o parâmetro "`RANDOM`" e gerado um valor pelo próprio protótipo.

6.3 Arquitetura da ferramenta FICTA

A ferramenta é composta por dois componentes: um aspecto e um arquivo XML. O aspecto contém os *advices* que possuem os códigos instrumentados com a lógica de injeção de falhas. O arquivo XML define os *pointcuts* e quais *advices* devem ser executados.

Cada tipo de falha considerada no modelo de falhas é implementado no aspecto e para realizar a injeção são realizadas interceptação e instrumentação de primitivas de comunicação do protocolo UDP. As classes que implementam os aspectos são carregadas da mesma forma que classes Java normais. O contêiner AOP fica responsável por diferenciar classes e aspectos e identificar em que momento os aspectos responsáveis pela instrumentação dos *bytecodes* devem executar no lugar do original. No caso do *framework* utilizado, AspectWerkz, essa função é realizada através da alteração do mecanismo de carga de classes do sistema para que seja possível essa identificação, conforme pode ser observado na Figura 1.

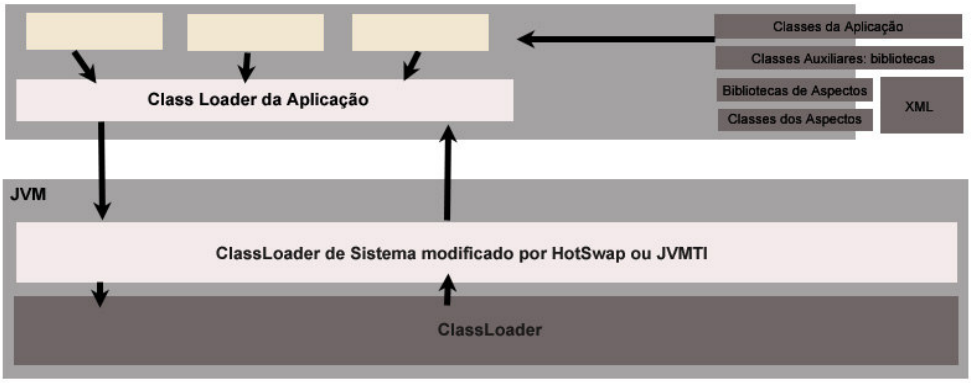


Figura 1 – Arquitetura de FICTA – ClassLoader de sistema modificado

Na integração de classes e aspectos, não é realizada uma instrumentação total da classe `java.net.DatagramSocket`, mas sim, são inseridos ganchos para a sinalização dos métodos que podem vir a ser instrumentados. Não é o *advice* com a lógica de injeção de falhas que é integrado na classe, mas sim o campo `JoinPoint`, definindo o *join point* especifico e uma chamada de método ao método `proceed()` da classe `JoinPoint`. Quando a classe alvo é instanciada, no caso a classe `java.net.DatagramSocket`, a instância do `JoinPoint` se registra no sistema e são inseridos os ganchos para ligar os *advices* aos *join points* que os definem.

6.4 Operação da ferramenta

A classe original `DatagramSocket` executa normalmente até que a aplicação alvo realize uma chamada a algum método que deva ser instrumentado. O contêiner AOP (*framework* AspectWerkz) identifica o *join point* (através do arquivo de definição em XML), recolhe informações sobre o mesmo, determinando quais os advices que estão ligados a ele e então passa o fluxo para o aspecto que implementa o advice responsável pela instrumentação do método. O *advice* verifica as condições que devem ser respeitadas para a injeção da falha através do método chamado `trigger()`. Esse método é privado do aspecto e realiza as verificações necessárias para definir se a falha deve ser injetada naquele momento ou não. Os cálculos são realizados a partir do número de mensagens que já foram enviadas ou recebidas pelo processo e também é levado em consideração se devem ser injetadas falhas de omissão ou colapso. Esses parâmetros são definidos em um arquivo de configuração. Após a execução do advice (método instrumentado), o fluxo é devolvido para o método original através do método proceed() da classe JoinPoint.

As falhas que devem ser ativadas durante a execução da aplicação, que compõem o cenário de falhas ou *faultload*, são especificadas em um arquivo texto. Esse arquivo é carregado juntamente com as classes e aspectos e as configurações são armazenadas em variáveis membros do aspecto. A Figura 2 ilustra um exemplo de como o arquivo pode ser configurado.

```
#parameters
perc=50
type=OMISSION
expType=DETERMINISTIC
```

Figura 2 – Exemplo de arquivo de configuração de FICTA

7 Aplicando FICTA em alvos reais

É importante ressaltar que esse artigo não visa uma validação de algum sistema específico, mas sim, o uso de algumas aplicações alvo como auxílio na prova de conceitos da abordagem apresentada.

Para demonstração da funcionalidade da ferramenta, foi escolhida uma aplicação distribuída cujo comportamento sob falhas fosse de fácil visualização. A mesma aplicação foi usada para demonstrar o uso da ferramenta FIONA [JAC2004a]. A aplicação alvo consiste de um quadro de anotações distribuído, onde cada vista do quadro reside em um diferente nó do sistema. Cada usuário pode fazer anotações na sua vista, essas anotações são distribuídas para todos os demais nós ponto a ponto. Se a aplicação não usa nenhum recurso de detecção e retransmissão de mensagens, uma mensagem omitida na origem representa um ponto em claro em todas as demais vistas, uma mensagem omitida no destino representa um ponto em claro apenas em uma vista. Um nó em colapso provoca a exclusão de participante no sistema, ou seja provoca uma alteração na visão de grupo.

Esse quadro de anotações distribuído usa o *middleware* de comunicação de grupo, JGroups. O JGroups permite que seja definida uma pilha de protocolos para que se atinja maior ou menor confiabilidade, de acordo com as necessidades do sistema que está sendo construído. Para a demonstração de FICTA, o quadro de anotações foi executado, definindo apenas o protocolo UDP na base de sua pilha de protocolos. No JGroups, o protocolo UDP é implementado como um invólucro para a classe `java.net.DatagramSocket` da API padrão Java. Dessa forma, FICTA atinge seus objetivos, que são injetar falhas instrumentando primitivas de comunicação.

Figura 3 – Experimento de injeção de falhas sem retransmissão

Para o primeiro experimento, definiu-se na pilha apenas o protocolo UDP, ou seja, nenhum mecanismo de tolerância a falhas. FICTA foi configurada para uma taxa de omissão de mensagens de 50%. A Figura 3 ilustra o protótipo agindo como esperado. A janela da esquerda corresponde ao processo onde as falhas estão sendo injetadas e onde o desenho está sendo construído. A omissão de mensagens é realizada no método `send` de `java.net.DatagramSocket`, impossibilitando o envio da mensagem para a rede, resultando na omissão da mensagem. A janela da direita possui um desenho incompleto, uma vez que as mensagens não estão sendo recebidas.

O mesmo teste foi realizado inserindo na pilha de execução do JGroups os protocolos que implementam os mecanismos de retransmissão e detecção de falhas. Quando um pacote UDP é perdido, é detectado pelos protocolos responsáveis, e então o pacote é retransmitido.

Os resultados desse teste mostraram que mesmo com falhas sendo injetadas, os desenhos em todas as janelas são idênticos. Esse comportamento é esperado, pois as falhas estão sendo tratadas. Com o auxílio desta aplicação alvo, foi facilmente visualizado que FICTA efetivamente injeta falhas de omissão e colapso. O quadro de anotações distribuído é um excelente caso de demonstração para ferramentas de injeção de falhas de comunicação.

8 Considerações Finais

Esse artigo apresentou uma nova abordagem para a construção de ferramentas de injeção de falhas baseada em programação orientada a aspectos. Também foi apresentada a ferramenta FICTA, um injetor de falhas, que utiliza a abordagem apresentada, e que tem como objetivo a validação de aplicações Java distribuídas construídas sobre o protocolo UDP e que implementam mecanismos de tolerância a falhas. O desenvolvimento da ferramenta mostrou que a abordagem é factível e eficiente e resulta em uma solução modular, reusável e sem intrusividade espacial nas aplicações alvo.

A opção pelo uso de aspectos surgiu ao se identificar a possibilidade de realizar a instrumentação do código da aplicação a ser validada sem a necessidade de alteração direta do código fonte, evitando a intrusividade do injetor de falhas no código do sistema alvo. Além disso, o uso de AOP trouxe ganhos em características como flexibilidade, modularidade e facilidade de manutenção e extensão. Quanto à flexibilidade, aspectos facilitam a introdução de funcionalidades de forma não pervasiva, permitindo que a arquitetura desenvolva-se sem intrusividade sobre o código. Quanto a modularidade, aspectos permitem o projeto de componentes modularizados de forma a evitar as desvantagens de espalhamento e emaranhamento de código. Finalmente quanto à facilidade de manutenção e extensão o código de injeção de falhas é mais fácil de manter e posteriormente estender para outros modelos de falhas, pois está localizado em um único lugar ao invés de espalhado pelo código.

FICTA pode ser aplicada em três cenários distintos para validar aplicações distribuídas com características de alta dependabilidade baseadas no protocolo de rede UDP. No primeiro cenário, FICTA é uma ferramenta de teste para o desenvolvedor da aplicação distribuída. O desenvolvedor testa sua aplicação para diferentes cenários de falhas, refinando-a caso a cobertura de falhas não seja satisfatória, sem a necessidade de alterar seu código e sem a preocupação de posterior exclusão de código extra de teste. Para esse cenário, FICTA está pronta para uso imediato. Em um segundo cenário o integrador de sistemas ou o responsável pela aquisição de software precisa avaliar se uma determinada aplicação de rede ou *middleware* para sistemas distribuídos, semelhante ao JGroups por exemplo, apresenta a cobertura de falhas desejada. O fornecedor do *software* não tem qualquer interesse em abrir seu código na fase de avaliação. Neste cenário, FICTA apresenta a vantagem de não necessitar do código fonte para instrumentar a aplicação e pode também ser imediatamente aplicada no estado em que se encontra. Em um terceiro cenário deseja-se medir a confiabilidade, disponibilidade, latência de erro, queda de desempenho sob falhas ou qualquer outra métrica de dependabilidade. Neste último cenário, FICTA pode compor um ambiente mais completo onde monitores, coletores e analisadores de dados estariam disponíveis para cálculo e análise das métricas.

FICTA é semelhante quanto a funcionalidade à extensão de Jaca e à FIONA. Entretanto, sua construção é completamente diferente, seguindo uma abordagem pioneira nesta área. Todas as três ferramentas prometem nenhuma intrusividade espacial. Um

interessante trabalho futuro é comparar as três ferramentas quanto a sua intrusividade temporal em sistemas alvo que apresentem restrições de tempo-real.

9 Referências

[ARL90] ARLAT, J. Et al. *Fault Injection for dependability Validation: a methodology and some applications*. IEEE Transactions on Software Engineering, v. SE-16, n.2, Feb.1990.

[BAN98] BAN, B. *JavaGroups – Group Communication Patterns in Java*. Department of Computer Science, Cornell University, July 1998.

[BON2004] BONÉR, J. VASSEUR, A. *AspectWerkz for Dynamic Aspect-Oriented Programming*. In Proc. Of AOSD 2004.

[CAR98] CARREIRA, J., LEITE, F.O., WEBER, T.S. *Building a Fault Injector to Validate Fault Tolerant Communication Protocols*. In Proc. of International Conference on Parallel Computing Systems. Ensenada, Mexico. Aug. 1998.

[DAW96] DAWSON, S. JAHANIAN, F., MITTON, T. *ORCHESTRA: A Probing and Fault Injection Environment for Testing Protocol Implementation*. In Procs. of IPDS'96. Sept. 1996.

[HSU97] HSUEH, M., TSAI, T., IYER, R. *Fault Injection Techniques and Tools*. IEEE Computer, v. 30, issue 4, April1997.

[JAC2004a] JACQUES, G., DREBES, R.J., GERCHMANN, J., WEBER, T.S. *FIONA*: *A Fault Injector for Dependability Evaluation of java-Based Network Applications*. In Proc. of 3rd IEEE NCA. Massachusetts, USA. 2004.

[JAC2004b] JACQUES, G., MORAES, R., WEBER, T., MARTINS, E. *Validando sistemas distribuídos desenvolvidos em Java utilizando injeção de falhas de comunicação por software*. V Workshop de Testes e Tolerância a Falhas. Maio 2004.

[KIC2001] KICKZALES, G. Et Al. An Overview of AspecJ. In Proc. 15th European Conference Object-Oriented Programming. Budapest, Hungary, June 2001.

[KIC97] KICKZALES, G. *Aspect Oriented Programming*. Proceedings of the European Conference on Object-Oriented Programming. June 1997

[LAD2002] LADDAD, R. *I Want my AOP!, Part 1 - Separate software concerns with aspect-oriented programming*. In Java World, http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect_p.html. 2002. Accessed in Feb. 2005.

[MAR2002] MARTINS, E., RUBIRA, C., LEME, N. *Jaca: A Reflective Fault Injection Tool Based on Patterns*. In Proc. of the 2002 International Conference on Dependable Systems and Networks. 2002.

[ROY2003] ROYCHOUDHURY, S., GRAY, J., WU, H., ZHANG, J., LIN, Y. *A Comparative Analysis of Meta-programming and Aspect-Orientation*. In Proc. Of the 41st Annual ACM SE Conference, Savannah, GA, March 2003.

Xception fault injection and robustness testing framework: a case-study of testing RTEMS

R. Maia[1], L. Henriques[1], R. Barbosa[1], D. Costa[1], H. Madeira[2]

[1]Critical Software SA
Parque Industrial de Taveiro, Lote 48, 3045-504 Coimbra, Portugal

[2]DEI-CISUC, University of Coimbra
Polo II, 3030-199 Coimbra, Portugal

{rmaia, lhenriques, rbarbosa, dino}@criticalsoftware.com,
henrique@dei.uc.pt

Abstract. *Xception is an automated and comprehensive fault injection and robustness testing environment that enables accurate and flexible V&V (verification & validation) and evaluation of mission and business critical computer systems and computer components, with particular emphasis to software components. In this paper we focus on the new robustness testing features of Xception and illustrate them with a concrete example of robustness testing of the Real Time Executive for Multiprocessor Systems (RTEMS) performed under a European Space Agency (ESA) contract. To the best of our knowledge, this is the first time that robustness testing results for this real time operating system are presented. The testing revealed a significant number of critical flaws in RTEMS 4.5.0 and shows the effectiveness of Xception toolset.*

1. Introduction

Test and dependability evaluation of computer systems and components are complex tasks and the growing complexity of both the hardware and software tend to make them even more difficult. Even the e valuation of very specific mechanisms such as error detection and recovery methods, which are relatively minor parts of the overall dependability evaluation, is traditionally a challenging task. Furthermore, the notion that functionality and performance are related to dependability is an established facet, as many systems operate often in degraded modes or with reduced performance due to faults.

Given the complexity of the task, there is no single generalized approach for testing and evaluating dependability features. Instead, several methods have been used ranging from pure modeling and analytical techniques to simulation and experimental approaches based on fault injection and robustness testing. These methods essentially complement each other, and depending on the phase of the design process, some of these methods could be more adequate or easy to use than others.

In general, analytical modeling and simulation are used to support architectural decisions at design phase. Detailed modeling techniques are also used for the evaluation of the dependability measures of the computer prototypes or even computer systems in field operation, but in these cases modeling needs the support of experimental approaches such as fault injection and field measurement. The experimental techniques

are used in computer prototypes or actual systems for system/components verification and validation, and constitute a very effective way to assess the efficiency of fault-tolerance mechanisms and to obtain the detailed characterization of the behavior of the whole system (or specific components) in the presence of faults or stressful conditions.

Xception is a comprehensive set of tools for experimental dependability evaluation. The Xception family started ten years ago with the proposal of a very low intrusiveness SWIF (software implemented fault injection) tool specifically targeted for very complex processors [Madeira 95, Carreira 95]. Since then, Xception has evolved to a comprehensive framework for experimental test and evaluation for dependability mechanisms and features, including several fault injection methods for a variety of target systems and a complete set of modules to assist the execution of testing campaigns and analysis of results. Xception has been used in many research and industrial projects and is being marketed by Critical Software SA (www.criticalsoftware.com).

In this paper we present the new robustness testing features of Xception and illustrated these features with an actual case-study of robustness testing of the RTEMS. Next section briefly describes Xception framework. Section 3 focus the robustness testing features and section 4 presents detailed example of the type of robustness tests that can be done by Xception. Section 5 concludes the papers and briefly describes future features.

2. Xception overview

2.1. Hybrid SWIFI Xception core

Xception has started and is best known as a SWIFI tool [Carreira 98]. The basic idea of SWIFI consists of interrupting the application under execution in some way and executing specific fault injection code that emulates hardware faults by inserting errors in different parts of the system such as the processor registers, the memory, or the application code. In general, the errors inserted are intended to emulate hardware transient faults, such as the ones that cause bit flips.

The first SWIFI approaches (i.e., prior to Xception proposal) inserted software traps in the code to mark the points during the code execution where the faults should be injected. However, these initial approaches had limited flexibility and the fault models were too simplistic, especially because no events related to time or data manipulation could be used as triggers. An alternative was to execute the code in trace mode, but this had enormous intrusiveness.

The Xception fault injection methodology uses a hybrid approach instead of a pure SWIFI methodology. The idea is to use the advanced debugging and performance monitoring features existing in modern processors (i.e., specific hardware inside the target processor) to inject more realistic faults by software and to monitor the activation of the faults and their impact on the target system behavior in detail. This approach has become a de facto standard for later SWIFI tools, like FTAFE [Tsai 96], MAFALDA [Rodríguez 99], and GOOFI [Aidemark 01].

The breakpoint registers available in the processors play a particularly important role in Xception basic fault injection approach, as they allow the definition of many

fault triggers, including fault triggers related to the manipulation of data. The processor is running at full speed and the injection of a fault corresponds normally to the execution of a small exception routine.

The performance monitoring hardware inside the processor is also used to collect detailed information on the behavior of the system after the injection of a fault. Particularly, the combination of the trigger mechanisms provided by the debugging hardware and the performance monitoring features of the processor, allows to monitor other aspects of the target behavior after the fault with minimal intrusion. For example, it is possible to detect if a given memory cell was accessed after the fault or if some program function (e.g., error recovery routine) was executed. Another important aspect is that, because Xception operates very close to the hardware (at the exception handler level), the injected faults can affect any process running on the target system including the kernel. It is also possible to inject faults in applications for which the source code is not available.

The hybrid SWIFI engine of Xception directly emulates physical transient faults in internal target processor units, main memory, and other peripheral devices that can be accessed by software. The fault triggers are based on breakpoint registers and allow the injection of faults in practically all circumstances related to code execution, data manipulation, or timing. The fault types consist of bit manipulations, which is a widely accepted model for hardware faults. Xception has been enhanced with specific modules to support several target processors include PowerPC, Intel Pentium and SPARC based platforms running Windows and Linux OSs and several real-time systems such as LynxOS, SMX, RTLinux, and ORK.

2.2. Current Xception toolset

The Xception tool has evolved from the initial hybrid SWIFI version into a comprehensive toolset including other fault injection methodologies and the new robustness testing features (presented in this paper for the very first time).

The Xception family includes the following tools:

- The original Xception tool, based on hybrid SWIFI technology.
- The extended Xception tool with the fault injection extensions based on scan chain technology (**SCIFI**).
- The new robustness testing tool.
- The Easy Fault Definition (**EFD**) and Xception Analysis (**Xtract**) add-on tools.

All tools share a common and key component: a graphic user interface named Experiment Manager Environment (**EME**), providing the backbone for the whole family of tools, thus offering a consistent view to the user.

The Xception toolset may result in a variety of different configurations, depending on the type of target system addressed and on the dependability evaluation needs. In addition to the generic components (EME, EFD, and Xtract), the actual Xception configuration required for each concrete case is achieved through the use of a set of modules (plug-ins) that implement specific fault injection or robustness testing methodologies for the different types of target systems. Figure 1 shows the Xception components and plug-ins currently available, including the new robustness testing plug-ins.

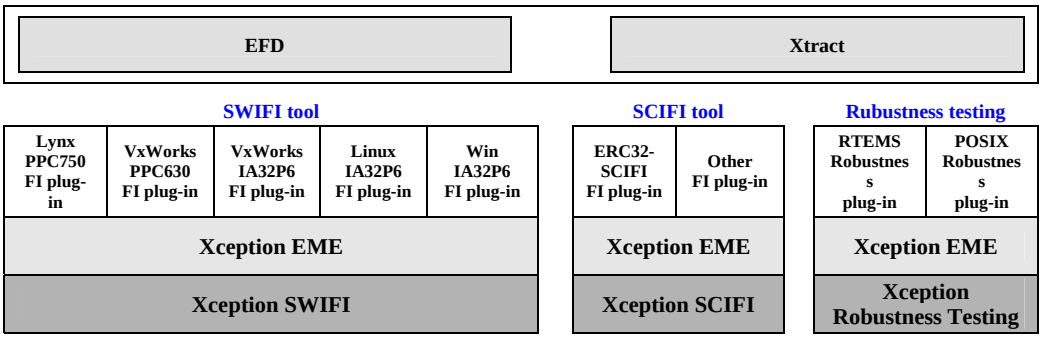


Figure 1. Xception components and plug-ins

The Xception architecture (see Figure 2) resembles the client-server model. It comprises a front-end module, which runs in a host computer and is responsible for experiment management/control (the EME components and, optionally, the EFD and Xtract components), and a lightweight injection core (FI hook) and monitoring elements (readout collector), which runs in the system under evaluation and is responsible for the insertion of the faults.

The connection between the host and the target is done by means of a high level protocol, built on top of TCP-IP. Experiment and fault configuration data flows from the host to the target, while fault injection raw data results flow in the opposite direction. The Xception tool includes all the components running on the host computer and the injection core and monitoring elements installed in the target system (see Figure 2).

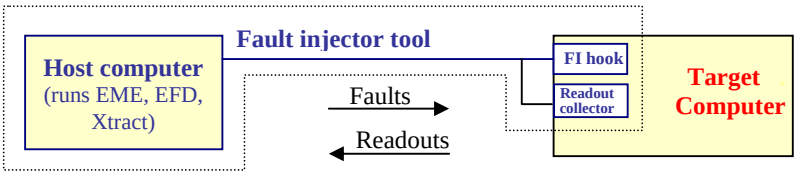


Figure 2. Xception components and plug-ins

The Xception EME (Experiment Manager Environment) running on the host computer is responsible for fault definition, experiment execution and control, outcome collection and basic result analysis. The EFD (Easy Fault Definition) is optional and is meant to facilitate the definition of faults through a high level view of the target system that allows the user to define faults triggers pointing directly to target code areas. The Xtract (Xception Analysis) is a stand-alone tool meant to perform detailed result evaluation, including detailed tracing of fault effects.

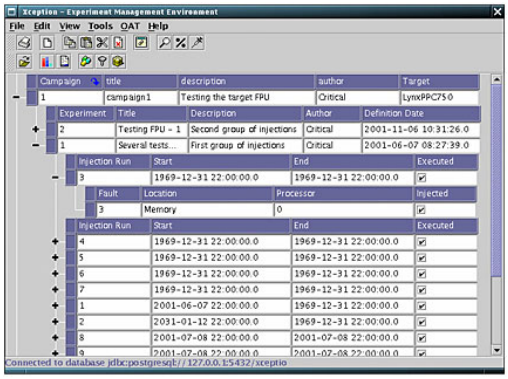


Figure 3. EME main window

Xception uses a relational database, enabling extensive outcome analysis. The Xception database holds all the information relevant to the fault injection and robustness testing process, including fault definition,

experiment execution and result analysis. The use of an open standard such as SQL (Structured Query Language) also allows the user to execute specific SQL queries or use other tools available in the market to explore the results, in addition to the tool already provided by the Xception.

3. Xception robustness testing methodology

The robustness testing methodology used by Xception is inspired on the robustness testing proposals from Carnegie-Mellon University [Koopman 97, Koopman 99] and LAAS [Rodríguez 99, Arlat 02]. The idea of classical robustness testing is to evaluate API (application programming interfaces) robustness in the presence of invalid inputs parameters. This idea has been applied to operating system calls but it can be generalized to inject faults in the interface parameters of any software component (we actually did that generalization in Xception robustness testing).

In robustness testing, interface faults change the input parameters turning them into invalid or exceptional values that may cause component failures. In principle, software components should behave in a robust way, even when they are submitted to invalid parameters (for example returning an error code). Unfortunately, most of the APIs are not robust, either because engineers assume that it is not practical to test and handle all the possible invalid inputs or because they just decide to overlook some input tests for performance reasons. The reality is that many API's are not robust as has been shown in several robustness testing works (e.g., [Koopman 99, Rodríguez 99]).

Once the input patterns that cause a given component to fail are identified, the problem can be solved either by changing the API, if the source code is available, or using wrapping when the source code is not available, which is normally the case of COTS components.

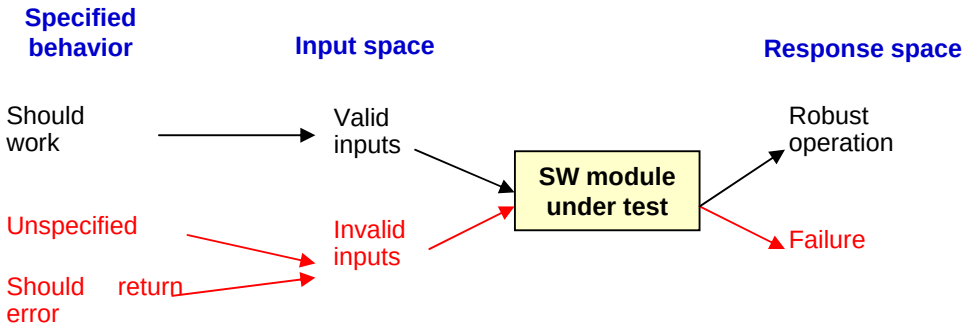


Figure 4 - Classical ([Koopman 99]) robustness testing approach.

Compared to classical approaches, Xception robustness testing approach includes some new aspects:

- It allows the inclusion of specific environment requirements (i.e., requirements imposed by other components of the target system or by the requirements external) in the robustness test cases. This includes, for example, timing requirements such as minimum response time in the presence of invalid input parameters.

- It is not specifically tied to operating systems or micro-kernel APIs (application programming interfaces) as it can be used to test the robustness of interfaces of any software component, including COTS (commercial off-the-shelf) and custom components. This generalization of robustness testing approaches is similar to Interface Propagation Analysis [Voas 98] and to the technique proposed in [Moraes 04].

  The Xception robustness testing methodology comprises three phases (see Figure 5):

- **Preparation**: Includes all the tasks needed to define test cases.
- **Test Execution**: Execution of the defined test cases.
- **Log Analysis**: Analysis of the results of the test cases and identification of the RTEMS faults.

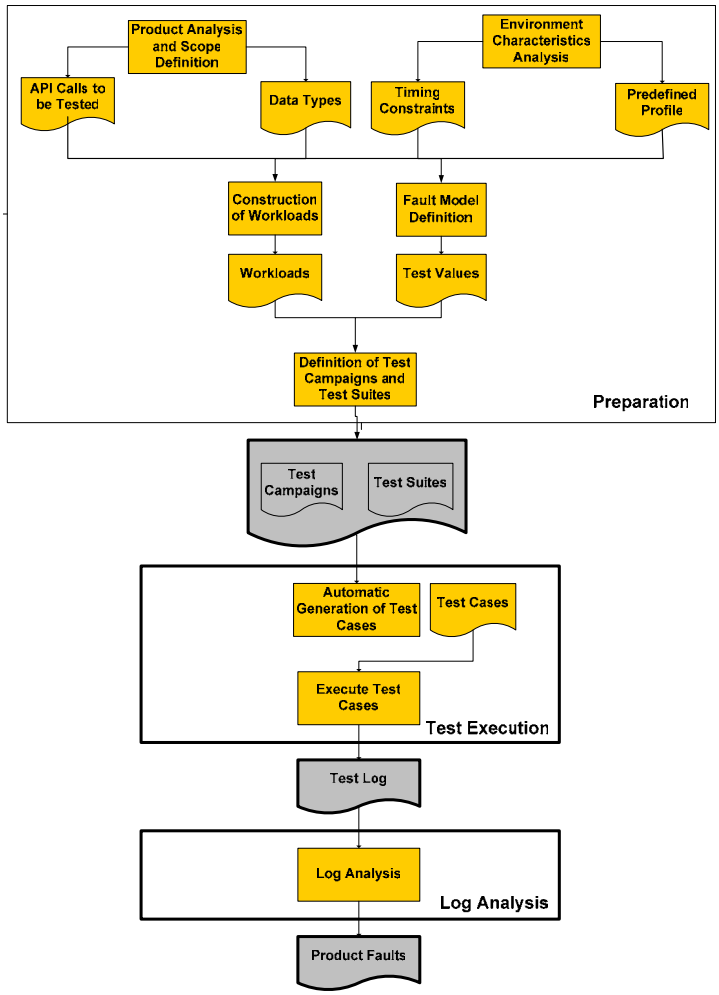


Figure 5 - Xception robustness testing methodology.

Preparation phase comprises the following tasks:

- Product Analysis and Scope Definition: Analysis of the product/component under evaluation and selection of the API calls that will be subjected to the evaluation.
- Environment Characteristics Analysis: Identification of extra requirements imposed by the environments (other target components and external environment) in the product/component under evaluation.
- Fault Model Definition: Definition of the in-bound and out-of-bound values that will be used as interface faults for each data type of the product/component under evaluation.
- Construction of the Workloads: Definition and implementation of the applications that will exercise the product/component under evaluation APIs.
- Definition of the Test Campaigns and Test Suites: Definition of the test suites that will be used to automatically generate the test cases (using the workload and fault model previously defined). Test suites are grouped logically in test campaigns.

The Test Execution phase follows the Preparation. During this phase test cases are executed and the results are collected to the Xception database. This task is performed in unattended mode by Xception.

The final phase of the robustness testing is the Log Analysis. In this phase detailed analysis of the log of each test case is performed comparing the obtained results against the expected values. This phase may be quite time consuming. For this reason it is important to have a concise workload output that enables the analyst to quickly find out if the result of the test case is consistent with the input parameters or not.

A very important aspect of robustness testing is the definition of the **interface faults** for each data type (see preparation phase described above). For each data type, a class of test values is defined in order to facilitate the definition of the test cases. For practical reasons, a test case is generated as mutants of the workload. Each mutant is the source code file of the workload that results from the application of a single mutation on a data parameter. These values differ from typical data values fed to functional tests (or unit tests) since they are specifically meant to exercise the error handling and robustness features of the component under test. These values typically reflect boundary or "magic" values in the data type range, or values that are semantic out-of-bounds in the scope of usage in a function call.

The test values (interface faults) for the basic data types were defined taking into account preparatory experiments and published data on robustness testing techniques, namely from Ballista project [Koopman 99]. The test values candidates are as follows (assuming C syntax):

- **Integers data types**: 0, 1, -1, MAX_INTEGER, MIN_INTEGER, selected powers of two, powers of two minus one, powers of two plus one.
- **Pointers data types**: NULL, -1 (cast to a pointer), pointers to free()'ed memory, pointers to malloc()'ed buffers of various powers of two in size.
- **Floats data types**: 0.0, 1.0, -1.0, ±MAX_FLOAT, ±MIN_FLOAT, PI and e.

In order to prevent an explosion in the number of test cases, but keeping good test coverage at the same time, a subset of the specified test values are normally selected. These test values do not represent all the possible values that may be interesting to test with the given data types. They have been chosen to provide a reasonable range of exceptional (non-nominal) input conditions to the software under test. Still, some rules apply. If the type to be used in a mutation is a pointer to a function, then the only possible value that will be used to create mutants is the NULL pointer. This decision is based on the fact that it is not interesting to pass invalid function pointers as parameters.

A final aspect (and also very important) is the failure mode classification. That is, when injecting interface faults in a given software component it may fail in several ways (failure modes). Xception has adopted the Ballista classification [Kropp 98] as a first step classification scale. This scale is known as CRASH and measures non-robust responses from the component under test. CRASH is an acronym for the following failures modes:

- **C**atastrophic failures: a complete system crash that requires a system reboot.
- **R**estart failures: the application hangs and requires application restart.
- **A**bort failures: abnormal termination of an application.
- **S**ilent failures: occur when it is reported that the called function or system call was completed successfully instead of returning an error indication (as it would be expected due to the invalid parameter values used in the call).
- **H**indering failures: incorrect error indication such as the wrong error reporting code.

As mentioned, Xception uses the CRASH scale only as a first step classification. In fact, as the impact of a robustness failure is highly dependent on the concrete application, in many cases is enough to identify robustness failures, without going into a detailed classification that may not apply to the concrete testing scenario. Another limitation of the CRASH scale is that it is assumes robustness testing of operating systems and is not adapted to other software components.

The natural alternative to the CRASH scale is to use the simplified scale suggested by Figure 4, which consider two possible responses: **robust operation** and **robustness failure**. In many cases it is useful (and easier) to classify robustness failures in critical and minor failures, according to the fault impact.

4. RTEMS case-study

This section presents a robustness testing study of the RTEMS 4.5.0 (Real Time Executive for Multiprocessor Systems) performed under a European Space Agency (ESA) contract.

4.1. RTEMS and experimental setup

Figure 6 shows the application architecture of RTEMS. It includes three sets of APIs: classic, POSIX and Itron. Only the first two API sets were tested in the present study.

The internal architecture for RTEMS can be regarded as a set of layers that work closely with each other to provide the necessary services to real time applications. The executive interface presented to the application is formed by grouping *directives* (API calls) into logical sets called resource managers. Scheduling, dispatching and object management are provided by the executive core, which depends only on a small set of CPU dependent routines.

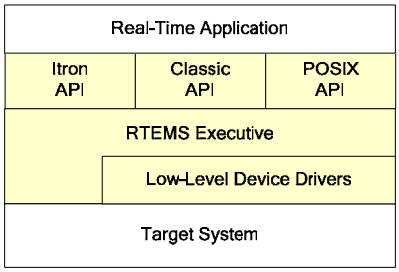


Figure 6. RTEMS Application Architecture

The Classic API provides seventeen resource managers as presented in Figure 7. For space reasons, we do not describe the resource managers as they represent traditional features provided by operating systems to real-time applications (a detailed description can be found in www.rtems.com).

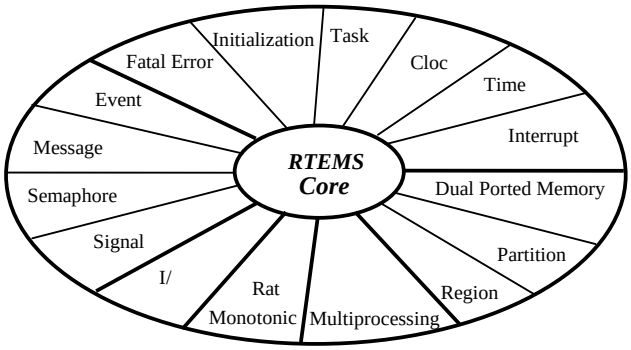


Figure 7. RTEMS Classic API resource managers

The POSIX API is based on the standard IEEE 1003.1b and provides nineteen managers, as shown in Figure 8.

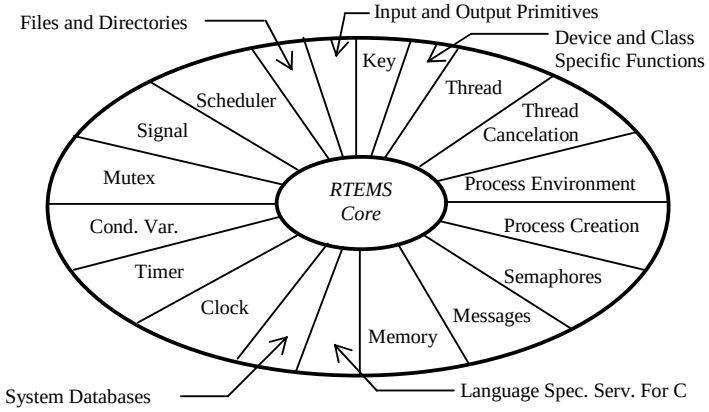


Figure 8. RTEMS POSIX API resource managers

The RTEMS executive core is responsible for the low level system management, including all CPU specific features and low level device drivers (see Figure 6). In the version tested, the CPU specific features belong to the SPARC/ERC32 processor. The RTEMS executive core also implements several handlers to be used by the API's to perform each specific function. These handlers include the message handler, the mutex handler, the semaphore handler, etc. The core was not designed to be used directly by the user's application, although no restrictions are imposed, at either compilation or linking time.

The experimental setup is the typical Xception setup, including the host computer and the target system, where the RTEMS is running (see Figure 9). For practical reasons, we have used a target system simulator instead of an actual board with the ERC32 processor. However, as the code executed in the target simulator is the actual RTEMS 4.5.0 code, the results are not affected by the fact the RTEMS is running in a processor simulator instead of an actual ERC32 board.

The Xception host is responsible for all the phases of the robustness testing experiment, as described in section 2. During the execution of the test cases, Xception creates the mutants following the test case definition. Then, the executable binary is built and the faulty application (mutant) is finally uploaded to the target system and executed. Data related to the execution is logged in order to allow the result analysis process.

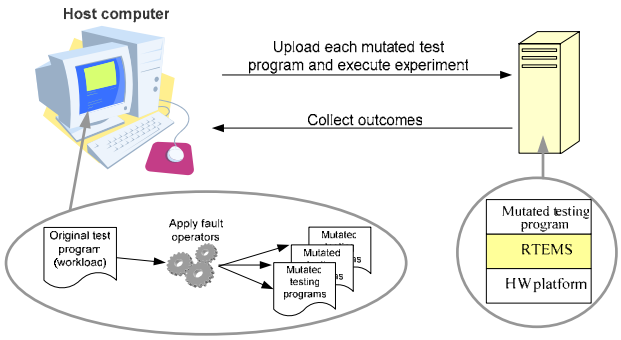


Figure 9. Experimental setup

4.2. Results

Tables 1 and 2 shows the summary of the results obtained for the resource managers already tested for the Classic API and the POSIX API, respectively. The first thing to note is the high number of robustness failures found in RTEMS 4.5.0, especially to what concerns critical failures.

We used the simplified classification of robustness failures mentioned in section 3 (instead of the CRASH classification). All the robustness failure situations have been manually analyzed and classified as **critical** or as **minor** failures, depending on the complexity of the recovery that would have been required in a real situation. In general critical failures include system and application crashes and minor failures include hindering cases. In most of the minor failures, RTEMS has returned a wrong error code.

The detailed analysis of the robustness failures cases has shown a variety of situations, all of them representing critical weak points of RTEMS. Note that all the failure cases

represent situations in which the RTEMS should have behaved in a robust way, for example returning a correct error code to the test program (instead of crashing or producing other unacceptable behavior). The following points show some examples of erroneous behavior observed:

- Unexpected change of the program control flow (e.g., when the task identifier is set to 0 on the rtems_task_start call).
- In several situations the test cases end up with unhandled traps such as *data access*, *memory not aligned*, and *illegal instruction* exceptions.
- Several situations where the RTEMS did not return any error code (silent failures). These cases are particularly dangerous, as they may affect the application in an unpredictable way.
- Several hindering failures where wrong error codes have been returned (i.e., instead of returning the error code specified in the documentation a different error code is returned).

Table 1. Classic API robustness testing results

Manager	Test Cases	Robustness failures		
		Critical	Minor	Total
Clock	68	0	0	0
Event	18	0	0	0
Fatal Error	3	0	0	0
Interrupt	5	0	0	0
IO	50	5	1	6
Message	83	2	6	8
Partition	27	0	2	2
Rate Monotonic	24	0	1	1
Region	67	4	3	7
Semaphore	33	0	1	1
Signal	10	0	1	1
Task	55	2	2	4
Timer	67	2	1	3
User Extensions	17	0	1	1
Total	**527**	**15**	**19**	**34**

Table 2. POSIX API robustness testing results

Manager	Test Cases	Robustness failures		
		Critical	Minor	Total
Clock	32	0	0	0
MESSAGE	122	2	1	3
MUTEX	223	1	3	4
Signal	122	1	4	5
Timer	29	0	3	3
Total	**528**	**4**	**11**	**15**

Due to space restrictions we cannot present a more detailed description of the results observed. The interested reader can find the full results in [Maia 04]. The detailed results also include code coverage results showing the lines of robustness testing code executed in each test case (zero lines in many of them) and the results of a comprehensive stress testing campaign (i.e., workloads that make extremely high usage of system resources).

4. Conclusion

This paper surveys the current features of the Xception fault injection and robustness testing family of tools. The main focus is on the new robustness testing features of Xception that are illustrated through a concrete case-study of testing the robustness of the RTEMS 4.5.0. To the best of our knowledge, this is the first time that robustness testing results for this real time operating system are presented. The results show a significant number of robustness failures situations (49 failures in total), including a large percentage of critical failures. These results clearly show that RTEMS should be improved for robustness before being used in critical applications.

References

Aidemark J., Vinter J., Folkesson P., Karlsson J., "GOOFI: Generic Object-Oriented Fault Injection Tool", Proceedings of Int. Conf. on Dependable Systems and Networks, DSN-2001, Göteborg, Sweden, 2001, pp. 71-76.

Arlat J., Fabre J.-C., Rodríguez M., and Salles F., "Dependability of COTS Microkernel-based Systems", IEEE Trans. on Computers, 51 (2) February 2002, pp 138-163.

Carreira J., Madeira H., and Silva J. G., "Xception: Software Fault Injection and Monitoring in Processor Functional Units", 5th IFIP Working Conference on Dependable Computing for Critical Applications, DCCA-5, Urbana-Champaign, Illinois, USA, September 27-29, 1995.

Carreira J., Madeira H., and Silva J. G., "Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers", IEEE Trans. on Software Engineering, 24 (2), pp.125-36, February 1998.

Koopman P. and DeVale J., "Comparing the Robustness of POSIX Operating Systems", in Proc. 29th Int. Symposium on Fault-Tolerant Computing (FTCS-29), (Madison, WI, USA), pp.30-7, IEEE CS Press, 1999.

Koopman P., Sung J., Dingman C., Siewiorek D., Marz T., "Comparing Operating Systems using Robustness Benchmarks", in Proceedings of the 16th International Symposium on Reliable Distributed Systems, SRDS-16, Durham, NC, USA, 1997.

Kropp, N., Koopman, P., and Siewiorek, D., "Automated Robustness Testing of Off-the_Shelf Software Components," 28th Fault Tolerant Computing Symposium, June 23-25, 1998.

Madeira H., Carreira J., and Silva J. G., "Injection of faults in complex computers", Fourth IEEE International Workshop on Evaluation Techniques for Dependable Systems, San Antonio, Texas, USA, October 2-3, 1995.

Maia R. et al, "RTEMS 4.5.0 Evaluation Report", CSW-RAMS-2003-CTR-1306, RAMS Call-off Order 2, ESTEC/Contract Nº 16582/02/NL/PA, 2004 (available on specific request to rmaia@criticalsoftware.com).

Moraes, R. and Martins, E. "An Architecture-based Strategy for Interface Fault Injection", Workshop on Architecting Dependable Systems, IEEE/IFIP International Conf. on Dependable Systems and Networks, Florence, Italy, June 28 – July 1, 2004.

Rodríguez M., Salles F., Fabre J.-C., and Arlat J., "MAFALDA: Microkernel Assessment by Fault Injection and Design Aid", in *Proc. 3rd European Dependable Computing Conf. (EDCC-3)*, (E. M. J. Hlavicka, A. Pataricza, Ed.), (Prague, Czech Republic), LNCS, 1667, pp.143-60, Springer, 1999.

Tsai T. and Iyer R. K., "An Approach to Benchmarking of Fault-Tolerant Commercial Systems", Proceedings of the 26th IEEE Fault Tolerant Computing Symposium, FTCS-26, Sendai, Japan, pp. 314-323, June 1996.

Voas J. and McGraw G. "Software Fault Injection: Inoculating Programs against Errors", John Wiley & Sons, New York, EUA, 1998.

Engineering a Failure Detection Service for Widely Distributed Systems

Bruno G. Catão , Francisco V. Brasileiro , Ana Cristina A. Oliveira

[1]Universidade Federal de Campina Grande
Coordenação de Pós-graduação em Informática
Laboratório de Sistemas Distribuídos
Av. Aprigio Veloso, 882, Bodocongó
58.109-970, Campina Grande, Paraíba, Brasil

{catao,fubica,cristina}@dsc.ufcg.edu.br

Abstract. *Unreliable failure detectors are recognized as important building blocks for implementing fault-tolerant distributed systems. Further, there has been a lot of discussion on how to provide them with sophisticated features that allow for adaptation, flexible use, scalability and quality of service enforcement. Despite that, we are not aware of any real distributed system that uses a sophisticated failure detection service. In fact, most systems deployed use the trivial failure detection scheme provided by the underlying communication technologies (e.g. TCP/IP timeouts). We believe that this state of affairs is due to two main reasons: i) there is no widely supported failure detection service API that incorporates these advanced features in a suitable way; and ii) the benefits of using a sophisticated failure detection service are not clearly understood. This paper targets the first issue by proposing a failure detection service that addresses the main necessities of widely distributed systems and implements the state-of-the-art in failure detection mechanisms. Moreover, to improve the usability of the service we took special care in the design of its programming interface.*

1. Introduction

In widely distributed systems it is normally not possible to establish an a priori upper bound on the end-to-end communication delays between processes. Therefore, failure detection in these systems is unreliable, since it is impossible to differentiate a crashed process from one that is simply running very slowly.

Nevertheless, in a seminal work, Chandra and Toueg proposed a theoretical framework to define the properties of useful unreliable failure detectors [Chandra and Toueg, 1996]. Following this theoretical work, various implementations of unreliable failure detectors were proposed (e.g. [Felber et al., 1999, Défago et al., 2003, Stelling et al., 1999, Hayashibara et al., 2004, Bertier et al., 2002]).

In addition, a lot of work has been devoted to extend the simple implementation sketched by Chandra and Toueg [Chandra and Toueg, 1996] and add more sophisticated features to a failure detection service. For instance, the first failure detection services proposed did not scale well. They generated too much control traffic when a large number of processes were monitored at the same time, degrading both the service as well as the network. Another problem is that these services were based on static timeouts. This leads to poor quality of service (QoS) in face of fluctuations in the execution environment. Further, earlier implementations did not provide

any guarantees concerning the QoS provided by the failure detection service. To address these problems, many approaches have been proposed in the literature. They provide ways to allow for scalability [Gemmell, 1997, Chu et al., 2000, Birman et al., 1999], adaptability [Chen et al., 2000, Bertier et al., 2002, Hayashibara et al., 2004], flexibility [Hayashibara et al., 2004] and QoS enforcement [Chen et al., 2000] to failure detection services.

Despite the many advances in the area, few (if any) real widely distributed systems use the sophisticated technologies developed so far. Instead, most systems seem to rely on very simple implementations based on the failure detection features provided by the underlying communication technologies (e.g. the default timeouts of TCP/IP sockets). We argue that two main reasons are responsible for this state of affairs. Firstly, since there is no widely supported ready-to-use failure detection service providing these functionalities, developers must cope with the cost of implementing the service from scratch. Secondly, the benefits of using a sophisticated failure detection service are not clearly understood. Therefore, it is not surprising that simpler solutions already available have been preferred.

As an attempt to reduce the gap between the state-of-the-practice and the state-of-the-art in failure detection for widely distributed systems, we propose the architecture of a failure detection service that encompasses sophisticated features in an easy to use way. Differently from previous works, our focus is both on the software engineering issues to develop a sophisticated and efficient failure detection service, as well as on the provision of a suitable programming interface for the service.

The remaining of the paper is structured in the following way. Section 2 specifies the requirements of a failure detection service for a widely distributed system. Section 3 summarizes the most suitable mechanisms proposed in the literature to implement the specified requirements. Section 4 discusses the service programming interface. Then, in section 5 we present the architecture of a failure detection service that efficiently accommodates the mechanisms discussed in Section 3. Section 6 shows how an application uses the service. Finally, Section 7 concludes the paper and presents some perspectives for future work.

2. Service Requirements

As discussed before, a failure detection service targeted for widely distributed systems must provide QoS guarantees at the same time that is scalable, adaptable and flexible. In this section we will explain in more details each one of these features.

Providing QoS guarantees is the ability of the service to be dynamically configured by applications to maintain a certain QoS level at runtime. The primary QoS metrics proposed by Chen et al. [Chen et al., 2000] are: detection latency, mistake recurrence time and mistake duration. The first metric is a random variable representing the total time that elapses from the time a process fails until its failure is permanently indicated by the failure detection service. The second metric is a random variable that represents the time between two consecutive mistakes (wrong suspicions of the failure detection service). Finally, the third metric is a random variable that represents the interval of time during which the failure detection service remains wrongly suspecting a process. Ideally, a client of the failure detection service should be able to define different upper and lower bounds[1] on the expected values of these metrics on a per-process basis.

A scalable failure detection service is able to monitor a large number of entities simultaneously without suffering a significant impact on its performance. The basic prob-

[1] Upper bounds for the first and third metrics, and lower bound for the second metric.

lem concerning scalability in widely distributed systems is how to transfer control messages from many sources to different distributed destinations without saturating neither the network nor the processing nodes, yet sustaining acceptable QoS levels.

Similarly, adaptability is the ability of the service to maintain an acceptable QoS even in face of changes in the execution environment. The environment over which widely distributed systems execute are very susceptive to these fluctuations. This is due to various factors, such as momentary increase on network traffic or CPU loads. A service designed to run in a widely distributed environment should be affected as less as possible by these fluctuations.

Finally, flexibility is the ability of the service to be usable by different applications. Conceptually, any service should be usable by the most different kinds of applications. In the context of a failure detection service this means that applications should be able to interfere in the task of deciding whether a process should be suspected or not. It is worth to point out that different decision requirements do not necessarily mean different QoS guarantees. A failure detection service for widely distributed systems must provide this flexibility level.

3. Providing the desired properties

Although the necessity of a failure detection service is evident, there are no ready-to-use failure detection service that provide all the desired properties described before. In this section we summarize the main contributions in the literature regarding the provision of these properties.

3.1. Providing scalability

To address this issue many works were developed. We can classify these works in two groups, the hierarchic and the epidemic solutions. The hierarchic solutions organize the entities in tree-based arrangements, in order to optimize the communication among the entities of the system [Ballardie et al., 1995, Chu et al., 2000, Jannotti et al., 2000]. On the other hand, the epidemic solutions work by diffusing its data using a random pattern that mimics the way infectious deceases spread themselves [Birman et al., 1999, Ganesh et al., 2001, Gupta et al., 2002]. Both solutions reduce the impact of the communication from an order of O(n^2) to an order or O($n \log n$), where n is the number of processes that monitor each other.

The main proposals to address scalability that relies on hierarchic approaches are based on multicast provided by the underlying network technologies (e.g. IP multicast) or data diffusion using overlay networks. The main advantage of the proposals based on native multicast technologies is that they are the most efficient way to transmit data from multiple sources to multiple destinations [Gemmell, 1997]. Other advantage is that all multicast functionalities are already implemented by the transport layer of the communication protocol. However, although there is more than a decade that such technologies have been released, the majority of routers still do not support them. Finally, another great disadvantage of these proposals is that some applications need QoS on the data diffusion, and native multicast technologies do not ensure this.

The main advantage of multicast trough overlay networks is that there is no necessity of a specific infrastructure. There is not even the necessity of a specific network protocol such TCP/IP. Other advantages are: it is possible to create specific solutions for each problem domain; the diffusion is transparent to the routers involved; the gain on scale due to the reduction of the amount of information stored and exchanged among routers;

and the simplified management. The main disadvantages of multicast through overlay networks are: bigger latency than in native multicast technologies, and the replication of messages in some links of the network.

Finally, a very interesting approach, that does not rely on explicit hierarchies, is the epidemic diffusion. Epidemic, or gossip, diffusion simulates the transmission of a infectious disease; an infected individual transmits its illness to other individuals in a randomized way. Besides being very simple, this method is also very effective. The first attempt to implement an epidemic protocol was done in the early 1980's. This protocol was developed for the USENET News Protocol. This protocol constructs a communication graph of a set of processes that should communicate. When a process wants to communicate with other processes it gossips its data to a subset of processes by choosing randomly some processes (at least $logn$ processes) and sending them the desired data. The processes that receive this data repeat this process several times, until reaching some probabilistic condition.

3.2. Providing adaptability

The main proposals to address adaptability in failure detection services are based on the use of dynamic timeouts. Following we will discuss the most important proposals.

One very important adaptable failure detector was proposed by Chen et al. [Chen et al., 2000]. Roughly speaking, this failure detector adapts itself by dynamically adjusting the timeout that each module sets when waiting for a heartbeat message (the lack of such message causes the service to start suspecting the process). This dynamic adjustment is done in a statistical way, sampling the heartbeat arrival times in order to estimate the future arrival times. A safety margin defined at deployment time is added to this estimated arrival time.

Another important adaptable failure detector was presented by Bertier et al. [Bertier et al., 2002]. Its adaptation mechanism is similar to the one shown in the previous solution with the difference that its safety margin is dynamically calculated using a mechanism based on the Jacobson's algorithm [RFC, 2000].

3.3. Providing flexibility

The first approaches intended to provide flexibility to failure detection services were based on the use of two timeouts [Defago, 2000, Defago et al., 1999, Defago et al., 1998]. By using these timeouts it is possible to configure the service to work at the same time with both applications that have conservative requirements (e.g. few wrong suspicions), as well as applications that have aggressive requirements (e.g. low detection latency).

The approaches based on two timeouts are useful, but they are not flexible enough for the wide range of requirements of widely distributed applications. To circumvent this problem, a new failure detector paradigm was proposed - the accrual failure detector [Hayashibara et al., 2003]. The main innovation introduced by the accrual failure detectors was the information provided by them. Traditional failure detectors export to applications binary information about the state of the monitored processes (suspected or correct). Accrual failure detectors, on the other hand, provide a probabilistic value associated with each monitored process, and this value indicates the confidence level about the liveness of the process.

3.4. Ensuring the quality of service

The proposals to ensure the QoS of a failure detection service are based on monitoring the QoS metrics, and, if required, adjusting the value of some control variables of the failure

detection service (e.g. the rate of heartbeat messages emission) so that it maintains its QoS as near as possible from an established set point [Chen et al., 2000].

In a heartbeat based implementation the heartbeat arrival times are stored in a sliding window of a pre-defined length, and the QoS metrics are calculated as a function of the values contained in this window. The length of the window is directly proportional to the accuracy of the QoS metrics and inversely proportional to the reactivity of these metrics. Also, the frequency of heartbeat sending, for each monitored process, is adjusted dynamically to meet the QoS requirements. For instance, if the detection latency is higher than the desired, the frequency of heartbeat sending is increased as a function of the difference between the desired and the measured detection latencies.

4. Service Programming Interface

Most of the papers that propose failure detection services pay little attention to the programming interface that applications use to have access to the service. They normally assume only a synchronous interface that allows clients to query the failure detection service for the status of a particular process. However, our experience developing real distributed systems [Our, 2005] shows that most of the time asynchronous interactions are preferred. This is because most systems are designed to act upon the occurrence of a failure. In other words, it is easier to program mechanisms for fault-tolerance by having them notified of the occurrence of a failure on a process, instead of having them constantly querying the failure detection service for the status of this process. This is also evident in the pseudo-code of many fault-tolerant algorithms presented in the literature (e.g. [Chandra and Toueg, 1996, Guerraoui and Raynal, 2005]). Of course, in some less frequent scenarios, synchronous interactions are also useful. For instance, before performing a costly task, it is worth querying the failure detection service to avoid the waste of resources (even though a query that returns that a process is correct gives no guarantees that it will remain correct in the near future).

In this section we propose an extended interface for a failure detection service that we believe is more useful for developers of fault-tolerant distributed systems. It provides both query-style and callback-style interactions. Different from previous proposals, it incorporates an asynchronous notification scheme that allows applications to register callback functions that are invoked whenever important events happen.

The architecture of the service proposed, shown in Figure 1, is based on the classic model that has three basic types of entities: monitors, monitorables and notifiables [Felber et al., 1999]. Monitor entities are responsible for probing the monitorable entities and notifying the notifiable entities whenever a failure happens. We extend this architecture by adding new relevant events, as described later in this section. Moreover, the monitor functionality is extended to allow for the monitoring of the QoS that is being provided, as well as the extra resource consumption due to the execution of the failure detection service.

There are four classes of events that may trigger a notification. The first class is related with the failure detection capabilities of the service. There are failure notification and wrong suspicion notification events. The second class comprises the QoS related notification events. They are used to notify applications when the current QoS levels attained by the service are below the QoS requirements defined by the application. There are three events in this class, one for each primary QoS metric (see Section 2). The third class comprises the events that are related with the impact that the service has on its execution environment. It has two events that are used to indicate, respectively, when the bandwidth consumption is above or below a given threshold. Finally, the fourth class is

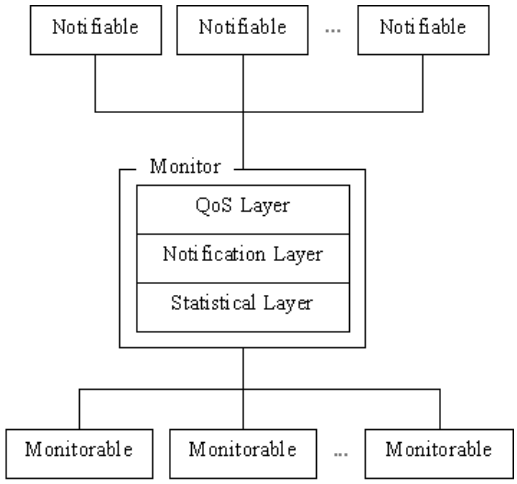


Figure 1: The main entities of the failure detection service.

related with configuration events. The single event of this class is used to notify notifiable entities of monitorable entities that are registered with some monitor entity. These events are useful for allowing notifiable entities to discover monitorable entities dynamically.

5. Implementing the service

In Section 2 we discussed the desirable properties that a failure detection service must possess. Then, in Section 3 we presented a summarized description of the most suitable mechanisms proposed in the literature to implement these properties. In this section we show how we packed together some of these mechanisms to implement a failure detection service that provides the programming interface described in the previous section.

The core of the failure detection service is implemented by the monitor entities. Monitorable and notifiable entities just implement simple interfaces to allow information to flow from monitorables to notifiables through a collection of monitors that cooperatively implement the service. The monitorable interface allows a monitor to probe a monitorable entity, while the notifiable interface allows a monitor to inform a notifiable entity that a relevant event has happened. This provides a lot of flexibility, since relevant events are specified on a per-entity basis. For instance, a given application may implement a notifiable entity that registers with a monitor to receive notifications of failures of a particular monitorable entity, whenever the probability of failure of the monitorable entity is above a particular confidence level.

To allow for adaptability and QoS enforcement it is important to have control on the rate with which control messages are exchanged between monitor and monitorable entities. Moreover, since the monitors are the entities responsible for controlling the behavior of the failure detection service, the communication among monitorable and monitor entities follows a pull style, i.e., the monitor entities send *"are you alive"* messages to monitorable entities, and monitorable entities respond to these messages by sending back *"I am alive"* messages. The pull communication model simplifies the control of the rate with which heartbeat messages are exchanged. If a push style were used, whenever a change in this rate was required, monitors would have to inform this to all monitorable entities.

The functionality of the monitor entities is implemented by three modules: the statistical module, the notification module, and the QoS module.

The statistical module receives control messages from monitorable entities and

computes two statistics about them: i) the probability that a process has failed, and; ii) the bandwidth generated by the failure detection service. It maintains a sliding window containing the size and the arrival time of each control message received. The probability that a process has failed is computed by calculating the average and the standard deviation of arrival times, using them to build a probability distribution and then, using its cumulative distribution function to obtain the probability value. By default the service uses a normal distribution, but other distributions can be used. The bandwidth generated by the failure detection service is calculated by adding the total of bytes received and dividing this by the time elapsed between the reception of the first and the last message in the window.

The notification module provides one of the interfaces that the notifiable entities have with the failure detection service. It allows the notifiable entities to specify the events whose occurrence will trigger the execution of a callback function. The other interface available to the notifiable entities is provided by the QoS module. It allows for both the definition of the desired levels of QoS, as well as the registration of callback functions that will be invoked whenever the current level of QoS attained is below the QoS requested.

Scalability was addressed in our implementation in an hierarchical way. Monitor entities were developed to behave also as notifiable entities. In this way, monitor entities could register with other monitor entities their interest in the state of some monitorable entities. Therefore, applications can build hierarchies of any number of levels following this pattern, as shown in Figure 2.

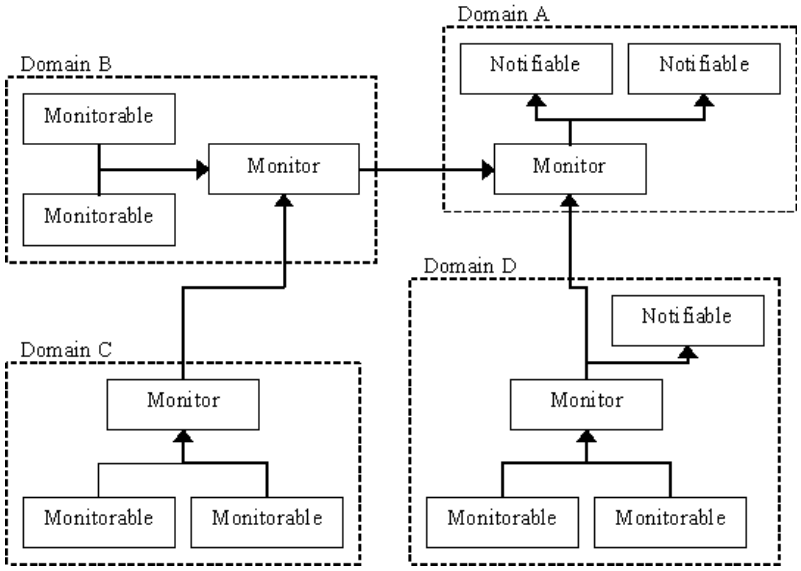


Figure 2: The monitors of the domain A, B, C and D compose a tree-based hierarchy; the notifiable entities of domain A could be registered as interested in monitorable entities of domain C.

To provide an asynchronous, transparent and simple way by which the entities of the system communicate, we have devised an event-based messaging service. This service is based on the concepts presented in [Starovic et al., 1995] and [Brasileiro et al., 2002].

The main entities of the messaging service are: Event Channel, Event Source, Event Listener, and Notification Context, as shown in Figure 3. The Event Channel hides from developers issues related to distributed communication. The Event Source entities are responsible for generating events and sending them to the event channel. The job of the Event Listener entities is to consume the events produced by the Event Source entities. Finally, the Notification Context is an abstraction of a domain within which the events exchanged by the entities will transit; this abstraction is very useful to reduce the network traffic delivering events only to the entities interested on them.

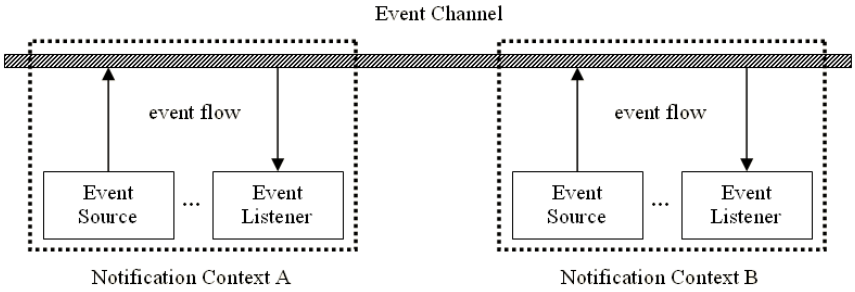


Figure 3: The entities of the messaging service.

It is worth to mention that the messaging service can be implemented in several ways. Our implementation uses Jini [JIN, 2005] as the communication technology. Another possibility is to use a gossip-style communication technology. We note that such an implementation would provide an alternative way to address the scalability issue in a more transparent way.

6. Using the Service

Figure 4 shows the interfaces of the main components of the failure detection service implemented.

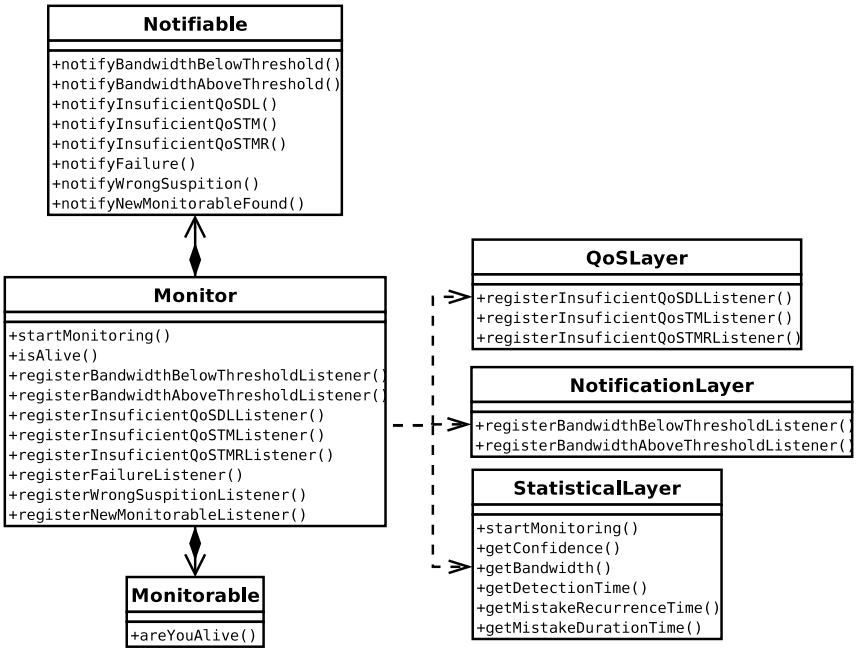


Figure 4: Simplified class diagram of the failure detection service displaying its main entities.

To use the service, applications must firstly obtain a monitor instance and then register with it the required monitorable and notifiable instances. A monitorable instance must be associated with every entity of the system that should be monitored. A notifiable instance must be associated with every entity of the system that is interested on the occurrence of relevant events. The actual monitoring is started by invoking the *startMonitoring* method on the appropriate monitor. For each pair of notifiable and monitorable entities a call to *startMonitoring* must be issued. By doing that, the invoker can define distinct QoS levels for each pair of notifiable and monitorable entities.

The process of locating and publishing entities in the failure detection service is accomplished through the messaging service. It defines primitives to manage the mem-

bership of the entities of the service. For every new instance of the service, the developers must call the *publish* method of the messaging service, passing to this method a name that uniquely identifies this instance in the system. The instances of the failure detection service can be searched by either their unique names or by their interfaces. In both cases the developers must call the *lookup* method, passing as a parameter the desired name or interface. The functionality of searching entities based on the interface is in fact a dynamic resource discovery mechanism. By using this feature, a monitor can, for instance, discover all the monitorable entities that are registered in the messaging service and start monitoring them, with no a priori knowledge about them. This is an alternative to using the configuration events described in Section 4. The sequence diagram in Figure 5 shows the steps that an application should take to use the failure detection service.

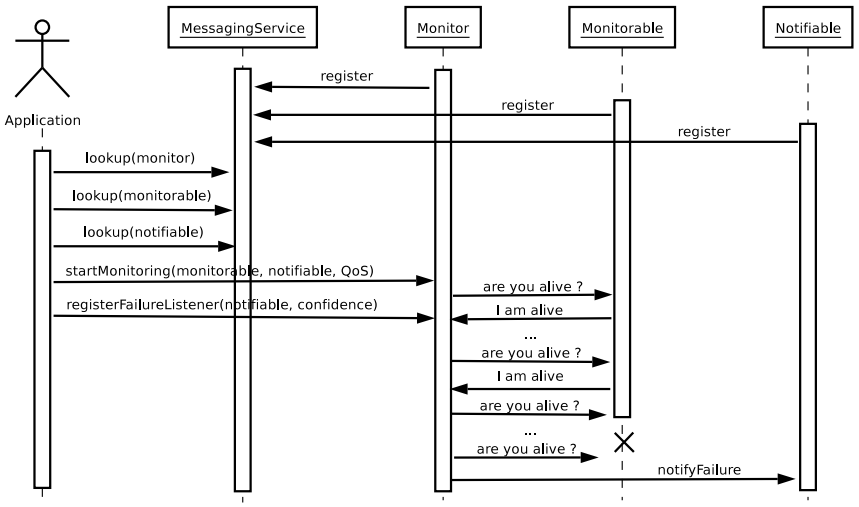


Figure 5: Sequence diagram showing how an application could use the failure detection service.

The monitorable interface is implemented by defining its *areYouAlive* method. This method is called by the monitor entities with which the monitorable is associated. A trivial implementation for this interface is to have the *areYouAlive* method always returning the *Monitorable.I_AM_ALIVE* value.

To implement the notifiable interface, developers must code the method that will be triggered by the monitor. The notifiable interface defines several methods that applications should implement according to their needs. See the notifiable interface in Figure 4.

The process of registering monitorable entities within a monitor entity is done in the following way. Firstly, it is necessary to obtain an instance of the monitor interface. Then, the application can call the *registerMonitorable* method of the monitor instance, passing to this method the required level of QoS and an instance for the corresponding monitorable entity.

The process of registering notifiable entities with a monitor entity is done in the following way. Again, the first step is to obtain an instance of a monitor entity. Then, it is possible to call one of the monitor's register methods. Applications must pass to these methods the parameters that specify when a notification should be triggered (e.g. the level of confidence that triggers a suspicion) and an instance for the correspondent notifiable entity.

The notification mechanisms are normally used as follows. First an appropriate QoS level is defined for each pair of notifiable and monitorable entities. This is indicated by invoking the *startMonitoring* method on the appropriate monitor passing as parameters the references for the notifiable and monitorable entities, as well as the QoS level

required. Since the QoS required may not be enforced by the failure detection service (for instance, because of the current load conditions of the environment), it is advisable that notifiable entities register to be notified whenever one of the QoS metrics is violated. When such a notification is received, the application may configure new QoS levels. For instance, sustain the level of mistake duration by augmenting the minimal detection latency required. Also, to balance acceptable QoS levels and resource usage, one can define lower and upper bounds on bandwidth consumption and register notification actions that are triggered whenever one of these thresholds are surpassed. When the upper limit on bandwidth consumption is signalled, the application may want to define less restrictive QoS levels in order to reduce control message traffic. On the other hand, when the resource consumption falls below the lower limit defined, application may want to increase its QoS level requirements.

Now we discuss the use of the failure detection service in the context of a particular example. Let us suppose that one wants to implement a service for supporting the installation, configuration and management of distributed applications. Such a system may be implemented by a centralized coordinator that executes at the management node and a number of distributed agents that run at the deployment nodes. The coordinator is responsible for instructing the appropriate agents to install a particular distributed application in all nodes that must run part of the application. Each agent is responsible for the installation of the part of the application that must execute at the agent's node. Moreover, a failure detection service is used for detecting the failures of both agents and applications, as well as crashes at the remote nodes. For simplicity, we assume that the coordinator does not fail. Whenever the part of the application or the agent that is running at a remote node are suspected, the coordinator is notified and takes the necessary actions to restart the application or the agent. Likewise, when a node fails, the coordinator chooses a new node to restart a new agent and the part of the application that was running at the faulty node. The failure detection service is used to notify the coordinator about failures on applications, agents and nodes.

To set the failure detection service up, first the coordinator instantiates a local monitor entity and registers itself to be notified of any monitorable entity that registers with this monitor. Whenever the coordinator is notified about a new monitorable, it registers itself with the monitor to be notified of the failures suspicions associated with this monitorable entity (with a confidence level that the coordinator indicates). Further, the coordinator starts the monitoring of the monitorable entity by invoking the *startMonitoring* method on the local monitor, passing as a parameter the required QoS level. Whenever a failure suspicion occurs, if the monitorable entity is associated with an application, then the coordinator contacts the corresponding agent, instructing it to reinstall the appropriate part of the application. On the other hand, if the monitorable entity is associated with an agent, then the coordinator may take one of two actions: i) if the application that runs on the same node of the agent is not suspected of having failed, then the coordinator will try to reinstall the agent at that node; ii) if both agent and application are suspected, then the coordinator assuming that the node has crashed, chooses a new node to start the agent and then tries to reinstall the application on this node.

At the distributed nodes, the failure detection service is set up in the following way. Each agent instantiates a local *Sensor*[2] that implements the monitorable interface on behalf of the agent. This *Sensor* locates the appropriate coordinator's monitor and registers itself with it. When an agent installs an application at its node, it instantiates a new *Sensor* to implement the monitorable interface on behalf of the application. This

[2] An auxiliary class *Sensor* that already implements the monitorable interface for generic Java objects is provided with the failure detection service API.

monitorable entity is also registered with the coordinator's monitor.

In this example the monitor that runs at the coordinator's node is responsible for probing all monitorable entities. If the number of monitorable entities is high, this may lead to a bottleneck. A trivial solution for this problem is to implement a hierarchy of monitors as discussed in Section 5.

7. Conclusion

Despite the many advances in the area of failure detection, few (if any) real distributed systems use failure detection services that implement the sophisticated mechanisms proposed in the literature. We believe that one of the causes of this state of affairs is the unavailability of a ready-to-use service that provides the appropriate API for most distributed applications. This paper tries to fill in this gap by presenting the architecture and the implementation of such a service[3].

As future work, we intend to provide alternative implementations of the messaging service based on gossip-style protocols. Further, we are integrating the service into the OurGrid[4] system [Our, 2005], a distributed grid middleware to support the execution of *bag-of-task* parallel applications. Our intent is to evaluate the benefits that a sophisticated failure detection service can bring to real distributed systems. We believe that the lack of perception of these benefits is another impairment to the broad utilization of services such as the one presented in this paper.

Acknowledgements

This work was partially developed in collaboration with HP Brazil R&D. Francisco Brasileiro would like to thank the financial support from CNPq/Brazil (grant 300.646/96).

References

(2000). Computing tcp's retransmission. N. W. Group. Rfc 2988. http://www.rfc-editor.org/rfc/rfc2988.txt.

(2005). The Jini Community. Sun Microsystems. http://www.jini.org.

(2005). The OurGrid Project. http://www.ourgrid.org.

Ballardie, T., Francis, P., and Crowcroft, J. (1995). Core based trees (cbt): An architecture for scalable multicast routing. In *ACM Sigcomm*, pages 88–95, San Francisco, USA.

Bertier, M., Marin, O., and Sens, P. (2002). Implementation and performance evaluation of an adaptable failure detector. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 354–363. IEEE Computer Society.

Birman, K. P., Hayden, M., Ozkasap, O., Xiao, Z., Budiu, M., and Minsky, Y. (1999). Bimodal multicast. *ACM Transactions on Computer Systems*, 17(2):41–88.

Brasileiro, F. V., Greve, F., Hurfin, M., Narzul, J. P. L., and Tronel, F. (2002). Eva: an event-based framework for developing specialised communication protocols. In *IEEE International Symposium on Network Computing and Applications*, pages 108–119.

[3]The software is available for download at http://www.spotter.lsd.ufcg.br/.

[4]Available at http://www.ourgrid.org/.

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. K. (2000). On the quality of sevice of failure detectors. In *International Conference on Dependable Systems and Networks (DSN'2000)*, pages 191–200, New York, USA.

Chu, Y.-H., Rao, S. G., and Zhang, H. (2000). A case for end system multicast. In *Measurement and Modeling of Computer Systems*, pages 1–12.

Defago, X. (2000). *Agreement-Related Problems: From SemiPassive Replication to Totally Ordered Broadcast*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland. Number 2229.

Defago, X., Felber, P., and Schiper, A. (1999). Optimization techniques for replicating corba objects. In *4th Int'l Workshop on Object-oriented Real-time Dependable Systems (WORDS'99)*, pages 1–8, Santa Barbara, CA, USA.

Défago, X., Hayashibara, N., and Katayama, T. (2003). On the design of a failure detection service for large scale distributed systems. In *Proc. Int'l Symp. Towards Peta-Bit Ultra-Networks (PBit 2003)*, pages 88–95, Ishikawa, Japan.

Defago, X., Schiper, A., and Sergent, N. (1998). Semi-passive replication. In *Symposium on Reliable Distributed Systems*, pages 43–50.

Felber, P., Guerraoui, R., Défago, X., and Oser, P. (1999). Failure detector as first class objects. In *International Symposium on Distributed Objects and Applications (DOA)*, pages 132–141, Edinburgh, Scotland,.

Ganesh, A. J., Kermarrec, A.-M., and Massoulie, L. (2001). SCAMP: Peer-to-peer lightweight membership service for large-scale group communication. In *Networked Group Communication*, pages 44–55.

Gemmell, J. (1997). Scalable reliable multicast using erasure-correcting re-sends. Technical report msr-tr-97-20, Microsoft Research Center.

Guerraoui, R. and Raynal, M. (2005). The information structure of indulgent consensus. *IEEE Transactions on Software Enginnering*, 54(4):453–466.

Gupta, I., Kermarrec, A., and Ganesh, A. (2002). Efficient epidemic-style protocols for reliable and scalable multicast. In *IEEE International Symposium on Reliable Distributed Systems (SRDS)*, pages 180–189.

Hayashibara, N., Défago, X., and Katayama, T. (2004). The φ accrual failure detector. In *Symposium on Reliable Distributed Systems (SRDS'2004)*, pages 66–78, Florianópolis, Brazil.

Hayashibara, N., Défago, X., and Katayama, T. (2003). Two-ways adaptive failure detection with the φ-failure detector. In *Workshop on Adaptive Distributed Systems (WADiS03)*, pages 22–27.

Jannotti, J., Gifford, D. K., Johnson, K. L., Kaashoek, M. F., and O'Toole, Jr., J. W. (2000). Overcast: Reliable multicasting with an overlay network. pages 197–212.

Starovic, G., Cahill, V., and Tangney, B. (1995). An event based object model for distributed programming. In *OOIS (Object-Oriented Information Systems) '95*, pages 72–86, London. Springer-Verlag.

Stelling, P., DeMatteis, C., Foster, I. T., Kesselman, C., Lee, C. A., and von Laszewski, G. (1999). A fault detection service for wide area distributed computations. *Cluster Computing*, 2(2):117–128.

Estendendo FIONA para uma arquitetura híbrida*

**Felipe Mobus, Roberto Jung Drebes, Gabriela Jacques-Silva,
Taisy Silva Weber, Ingrid Jansch-Pôrto**

Instituto de Informática – Universidade Federal do Rio Grande do Sul
Caixa Postal 15064 – 90501-970 Porto Alegre, RS, Brasil

{fmobus,drebes,gjsilva,taisy,ingrid}@inf.ufrgs.br

Abstract. *This papers discusses the importance of fault injection as a tool for the validation of distributed applications and points how to integrate two fault injection tools for better interoperability in a hybrid distributed fault injection architecture.*

Resumo. *Este trabalho discute a importância da injeção de falhas para a validação de aplicações distribuídas e aponta como integrar duas ferramentas de injeção de falhas para obter uma melhor interoperabilidade em uma arquitetura híbrida de injeção de falhas distribuída.*

1. Introdução

A crescente uso de sistemas distribuídos exige que estes sejam dotados de mecanismos apropriados de tolerância a falhas, garantindo seu funcionamento seguro. Aplicações distribuídas estão sujeitas a distúrbios de transmissão, podendo sofrer inconsistências e tendo seu comportamento imprevisível. É necessário garantir que estas aplicações tenham mecanismos de tolerância a falhas adequados antes de colocá-las em ambientes de produção. Para isto, o uso de componentes de boa qualidade e de arquiteturas de sistemas bem planejadas não é suficiente: o mal funcionamento de mecanismos de detecção e recuperação pode acarretar em resultados desastrosos [1].

A injeção de falhas emula a ocorrência de falhas que, de outra forma, ocorreriam com freqüência muito baixa, assim facilitando a observação de seus efeitos sobre o sistema, apontando seus gargalos de dependabilidade e determinando a cobertura dos mecanismos de detecção e recuperação de falhas [2]. Ao desenvolver injetores de falhas, é mais adequado usar injetores genéricos ao invés de desenvolver um injetor específico para cada aplicação, visando diminuir o custo associado à fase de validação.

Este trabalho demonstra a possibilidade de estender o injetor de falhas de arquitetura distribuída FIONA (*Fault Injector Oriented to Network Applications*) [3] de forma a torná-lo interoperável com outros injetores de falhas. Como exemplo desta extensão é utilizado o injetor ComFIRM (*Communication Fault Injector through OS Resource Modification*) [4]. A Seção 2 apresenta as duas ferramentas de injeção de falhas e suas abordagens distintas. Em seguida, é discutida uma forma de integrá-las para a injeção distribuída de falhas. O artigo encerra com algumas conclusões sobre esta integração.

*Desenvolvido em parceria com HP Brasil P&D e Projeto CNPq ACERTE (#472084/2003-8)

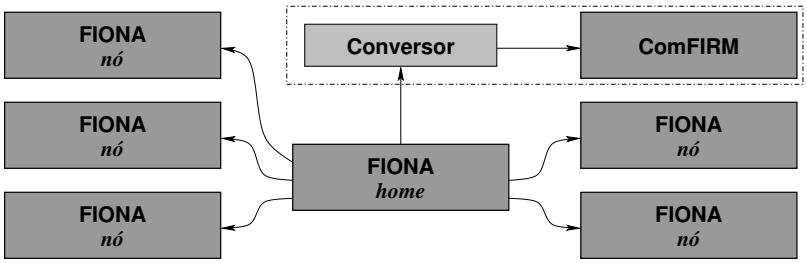


Figura 1: Modelo de integração do FIONA com outros injetores de falhas

2. Os injetores FIONA e ComFIRM

Neste artigo, são abordadas duas ferramentas de injeção de falhas distintas: FIONA e ComFIRM. FIONA é um injetor de falhas para aplicações Java distribuídas de larga escala. Sua abordagem para injeção de falhas é a instrumentação das classes de comunicação através de interfaces nativas de depuração da plataforma Java. A ferramenta permite a experimentação coordenada de injeção de falhas ocorridas em nós distintos. Para isto, uma instância de FIONA assume o papel de máquina *home* do experimento, e as demais tornam-se clientes de uma arquitetura distribuída, como injetores de nó.

ComFIRM, por sua vez, é destinado a aplicações para a plataforma GNU/Linux e funciona através da inspeção e manipulação de pacotes diretamente no interior do núcleo do sistema operacional. ComFIRM não apresenta arquitetura distribuída para injeção de falhas, mas, como mostrado a seguir, a ferramenta pode ser integrada à arquitetura FIONA como injetor de nó para aplicações não Java.

3. Estendendo FIONA para novos injetores de nó

FIONA é um injetor aplicável a sistemas cujos nós todos executem aplicações Java. Para usá-lo em sistemas híbridos, como um servidor de plataforma Linux e clientes em Java, é necessário estender FIONA permitindo o uso do injetor de falhas de nós distintos, compatíveis com as aplicações alvo. Se a integração for feita de forma transparente à máquina *home*, convertendo as especificações dos cenários de falhas nos nós da arquitetura distribuída e usando o mesmo protocolo de comunicação que um nó FIONA, a necessidade de modificação da ferramenta será mínima.

Para isto, foi desenvolvido um módulo responsável pela comunicação via o protocolo de um nó FIONA e pela tradução das regras de configuração de experimento transmitidas pela máquina *home*. Estas são convertidas em regras e comandos com a mesma semântica mas sintaxe da ferramenta de injeção utilizada nos nós que não utilizam FIONA, como ilustrado na Figura 1. Usando esta abordagem, não são necessárias modificações nas ferramentas envolvidas, beneficiando a modularidade. Para demonstrar a aplicabilidade dessa abordagem, exemplificamos a integração da ferramenta ComFIRM como caso de estudo. Técnicas semelhantes podem ser utilizadas para outras ferramentas.

4. Integração ComFIRM-FIONA

Ao desenvolver o módulo de conversão, é necessário distinguir as falhas injetadas por FIONA que necessitam de coordenação, como particionamento, das demais falhas.

```
teste  pacote.endereço == conexão.endereço
teste  pacote.porta == conexão.porta
teste  conexão.flag == 0
ação   conexão.flag := 1
ação   conexão.contador := 0
ação   conexão.temporizador := 0
ação   conexão.temporizador.modo := incrementar
```

Figura 2: Regra de inicialização

No primeiro caso, é preciso traduzir a mensagem de coordenação enviada aos nós envolvidos para comandos da ferramenta ComFIRM que produzam o comportamento esperado para o nó. No outro caso, é necessário traduzir do arquivo de parâmetros do cenário de falhas enviado ao nó para o formato de entrada de parâmetros exigido por ComFIRM.

4.1. Tradução dos cenários de falhas

ComFIRM analisa os pacotes de todas as aplicações em execução no sistema operacional. Como normalmente se deseja injetar falhas apenas em determinadas aplicações, é preciso verificar se os pacotes pertencem à aplicação alvo, testando os endereços de *socket* e só realizando a injeção se o *socket* pertence à aplicação em teste.

O teste de endereço pode ser especificado com poucas instruções da ferramenta ComFIRM. Ele consiste da leitura de campos específicos do pacote analisado e é fundamental para a seleção correta dos pacotes a serem tratados. Para traduzir para o ComFIRM o intervalo temporal de injeção de falhas, é utilizado, para cada regra traduzida, um contador e um temporizador, disponíveis no ComFIRM. Assim, para cada pacote da aplicação, é feito um incremento no contador e um teste de intervalo, ou seja, um teste do valor do contador/temporizador contra os limites inferior e superior do intervalo. Para a determinação da taxa de falhas, um valor aleatório é gerado por ComFIRM contra o qual também é feito um teste de limite. Sendo o pacote selecionado pelos testes acima descritos, a falha propriamente dita é inserida, de acordo com o tipo de falha indicada.

É necessário ainda criar uma regra de inicialização para cada falha traduzida, garantindo o início em zero dos valores do contador e do temporizador para a conexão. A construção de tal regra consiste em um teste de endereço para a conexão e um teste se o pacote atual é o primeiro da conexão, feito através de um *flag* referente a conexão. Em seguida, inicializa-se em zero o temporizador e o contador da conexão de forma que essa regra não seja novamente executada. A estrutura desta regra pode ser vista na Figura 2.

4.2. Coordenação de injeção de falhas

A emulação da ocorrência de falhas, como particionamento de rede, exige coordenação por se tratar de uma falha que afeta mais de um nó por vez. No caso deste tipo de falha, a máquina *home* envia a todos os nós a descrição do intervalo de ocorrência da falha e as máquinas envolvidas. A partir dessas informações, o módulo de conversão configura a ferramenta ComFIRM com as regras de injeção de falhas de colapso necessárias para causar o particionamento descrito pela máquina *home*.

Cada falha de colapso, para ser injetada, necessita apenas de uma regra de injeção, composta por um teste de endereço e uma ação de descarte, como mostrado na Figura 4.

```
teste  pacote.endereço == conexão.endereço
teste  pacote.porta == conexão.porta
teste  conexão.contador >= falha.começo
teste  conexão.contador <= falha.fim
ação   conexão.contador++
teste  rand() <= falha.taxa
ação   injeta.falha(tipo_de_falha)
```

Figura 3: Regra de Injeção de falhas genérica

```
teste  pacote.endereço == conexão.endereço
ação   injeta.falha(descarte)
```

Figura 4: Regra de colapso para particionamento

Essa regra deve ser adicionada no início do particionamento coordenado e removida ao final do mesmo. Adicionalmente, é necessário instruir a ferramenta ComFIRM a não filtrar pacotes utilizados para a coordenação das falhas, criando uma regra que permita a recepção dos pacotes destinados à porta utilizada pelo módulo de comunicação do injetor com a máquina *home*.

5. Considerações finais

Os injetores de falhas são importantes ferramentas de validação da dependabilidade, pois permitem a análise do comportamento das aplicações sem a necessidade de esperar pela ocorrência natural de falhas. Neste artigo, é apresentado um modelo de integração de injetores distintos de falhas de nó em uma arquitetura distribuída, aumentando a interoperabilidade.

O modelo de integração apresentado utiliza um módulo de comunicação e tradução de configuração do cenário de falhas do injetor de nó sem modificação da semântica descrita originalmente pela máquina *home* do experimento. A abordagem apresentada objetiva a manutenção da modularidade da arquitetura de injeção de falhas, se integrando transparentemente a arquitetura de FIONA sem exigir alterações dos injetores integrados.

Referências

[1] Carreira, J. e Silva, J. G. (1998) Why do some (weird) people inject faults? *ACM SIGSOFT Software Engineering Notes*, 23 (1): 42–43.

[2] Hsueh, M.-C., Tsai, T. K. e Iyer, R. K. (1997). Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82;

[3] Jacques-Silva, G., Drebes, R. J., Gerchman, J. e Weber, T. S. (2004). FIONA: A fault injector for dependability evaluation of Java-based network applications. In *Proc. of the 3rd IEEE Intl. Symposium on Network Computing and Applications*, páginas 303–308, Cambridge, MA.

[4] Leite, F. O. (2000). ComFIRM – injeção de falhas de comunicação através da alteração de recursos do sistema operacional. Dissertação de Mestrado, II/UFRGS, Porto Alegre, Brasil.

Testando uma Infra-estrutura FT-CORBA: O Caso GROUPPAC *

Alexandre Schulter[1] **, Alysson Neves Bessani**[1†]**, Eduardo A. P. Alchieri**[1] **,**
Joni da Silva Fraga[1] **, Lau Cheuk Lung**[2]

[1]DAS - Departamento de Automação e Sistemas
UFSC - Universidade Federal de Santa Catarina
Campus Universitário, Caixa Postal 476 - CEP 88040-900 - Trindade - Florianópolis - SC

[2]PPGIA - Programa de Pós-Graduação em Informática Aplicada
PUC-PR - Pontifícia Universidade Católica do Paraná
R. Imaculada Conceição, 1155 - Prado velho - CEP 80215-901 - Curitiba -PR

Abstract. *This paper presents a case-study about the application of unit tests in the* GROUPPAC *fault tolerance infrastructure. Our approach is pragmatic: given the system' dimensions and the current state of his development we opt for apply techniques and tools widely used, testing only the services' interfaces that compose the infrastructure. This approach allowed us to improve the overall quality of the system.*

Resumo. *Este artigo apresenta um estudo de caso sobre a aplicação de testes de unidade na infra-estrutura para tolerância a faltas* GROUPPAC*. Nossa abordagem é pragmática: dadas as dimensões do sistema e o estágio atual de seu desenvolvimento optamos por aplicar técnicas e ferramentas largamente utilizadas testando apenas as interfaces dos serviços que compõem a infra-estrutura. Esta abordagem nos permitiu melhorar a qualidade do sistema em geral.*

1. Introdução

A tolerância a faltas (TF) é um requisito de grande importância em aplicações críticas distribuídas. Em geral, esta qualidade de serviço é implementada através da utilização de algum suporte que implemente os protocolos e mecanismos necessários para que uma aplicação mantenha-se ativa mesmo com a falha de alguns de seus componentes. Estes suportes são comumente implementados em nível de *middleware*, fornecendo serviços básicos de TF à aplicação. Um requisito fundamental de qualquer suporte de TF é a confiabilidade e correção: se a aplicação deve tolerar faltas a infra-estrutura não pode conter erros que levem a falhas.

Dentre os padrões de *middleware* atualmente em uso, o CORBA (*Common Object Request Broker Architecture*) se destaca pela sua maturidade, sendo aceito nos mais diversos segmentos. Além dos mecanismos básicos de comunicação, interoperabilidade e serviços, este padrão contempla uma arquitetura para objetos distribuídos tolerantes a faltas (de parada), chamado FT-CORBA (*Fault-Tolerant CORBA*) [Object Management Group, 2002]. Este suporte é definido a partir de um conjunto de objetos de serviço usados no gerenciamento de objetos de aplicação que são replicados usando abstrações de grupos.

O GROUPPAC é um suporte para tolerância a faltas que implementa o padrão FT-CORBA [Lung et al., 2001]. Este suporte vem sendo desenvolvido no LCMI/DAS/UFSC desde 1999 e atualmente se encontra em sua terceira versão [Bessani et al., 2004].

Como em qualquer outro software concebido para ser utilizado em projetos reais, espera-se que o mesmo funcione corretamente. Este requisito se acentua se levarmos em consideração o ambiente para o qual o GROUPPAC foi desenvolvido: *software* crítico. Desta forma, uma melhora da qualidade do sistema pode ser obtida através da aplicação de testes de software.

*Financiado pelo CNPq (edital Software Livre), projeto GROUPPAC/MJACO (número 401802/2003-5).
†Autor para contato: *neves@das.ufsc.br*

O processo de teste é uma atividade básica do ciclo de desenvolvimento de software. Durante este processo um programa é executado (exercitado) com intenção de encontrar erros [Kaner et al., 1999]. O processo de testes não demonstra que um programa está correto, mas mostra seu comportamento considerando um conjunto de entradas pré-definidas (casos de teste). Se os casos de testes considerados forem de boa qualidade é possível ter uma boa estimativa da qualidade de um sistema e encontrar uma série de possíveis erros em seu código.

Este artigo apresenta uma breve descrição do processo de testes empregado para validar a implementação da infra-estrutura GROUPPAC. Este processo baseou-se na utilização de técnicas simples e largamente aceitas como testes de unidade [Beck and Gamma, 1998] apoiado por ferramentas. As próximas seções do artigo apresentam uma breve descrição da arquitetura do GROUPPAC (seção 2), a abordagem utilizada para testes e seus resultados (seção 3) e algumas considerações finais sobre o trabalho (seção 4).

2. GROUPPAC

O GROUPPAC [Lung et al., 2001] é uma implementação completa do padrão FT-CORBA desenvolvida inicialmente visando dar suporte a sistemas de larga escala. O objetivo básico desta infra-estrutura de TF é fornecer um conjunto de serviços úteis na construção de aplicações tolerantes a faltas. Sua arquitetura é composta por uma série de serviços, dentre os quais destacam-se:

- **Serviço de Gerenciamento de Replicação - SGR** Este serviço é responsável pelo ciclo de vida dos grupos de réplicas. Duas funcionalidades básicas são oferecidas pelo mesmo: **Gerenciamento de Propriedades**, onde as propriedades da replicação (tipo de replicação usada, número mínimo de réplicas, etc.) são definidas e alteradas; e o **Gerenciamento de Grupos**, que oferece mecanismos para o controle da criação e destruição de grupos, bem como de seus membros (réplicas). Sendo assim, responsável pela manutenção da filiação (*membership*) dos grupos gerenciados.
- **Serviço de Gerenciamento de Falhas - SGF** Envolve um conjunto de objetos que cooperam na detecção, notificação, análise e diagnóstico de faltas. Quando uma falha é detectada esta é repassada ao SGR para que este execute as ações necessárias e mantenha uma lista de membros sempre atualizado dos grupos.
- **Serviço de Gerenciamento de Recuperação e Logging - SRL:** Este serviço define mecanismos para a recuperação de réplicas e do próprio grupo, bem como da configuração de estados em membros que entram no grupo após o serviço replicado já estar em funcionamento. Entre estes mecanismos está um suporte para construção de *logs* usados no armazenamento de requisições enviadas ao grupo e um mecanismo para atualização de membros através de *checkpoints*.

Além destes serviços básicos, outros serviços como o de transferência de estados e o de comunicação de grupo, entre outros, também fazem parte da arquitetura do sistema. Entretanto, conforme será visto a seguir, os testes realizados se basearam diretamente no SGR, SGF e SRL.

3. Testes

Existe uma infinidade de abordagens e ferramentas desenvolvidas visando melhorar o processo de teste de software, grande parte destas são de cunho acadêmico e/ou experimental. Desta forma, nesse trabalho, optou-se por uma abordagem pragmática: dadas as dimensões do sistema e o estágio avançado de seu desenvolvimento optamos por testes que exercitam apenas as interfaces dos serviços que compõem a infra-estrutura verificando se os resultados obtidos estão de acordo com o esperado segundo as especificação do FT-CORBA.

Dentro desta abordagem essencialmente prática, optou-se pelo uso da ferramenta JUNIT [JUnit, 2005]: um *framework* para criação de testes de unidade [Beck and Gamma, 1998]. No entanto, os testes escritos caracterizam-se como testes de sistema, pois cada teste inicializa a maioria dos serviços do GROUPPAC de forma integrada e atua como um usuário da infra-estrutura executando diferentes casos de uso (chamando os métodos das interfaces) com entradas (parâmetros) corretas ou não. Estes testes estão divididos em quatro grupos, apresentados a seguir.

3.1. Fábrica Genérica

Uma fábrica genérica é um objeto utilizado pelo SGR para criação e remoção de réplicas (objetos) localmente nos *hosts* do sistema. Estes testes utilizam uma fábrica diretamente, sem interferência do Gerenciador de Replicação e verificam a criação de objetos CORBA passando parâmetros corretos e incorretos. No caso de parâmetros corretos, é verificado se o objeto foi realmente criado e o identificador retornado é correto. No caso de parâmetros incorretos, os parâmetros são passados incorretamente de diversas formas e o comportamento esperado é o lançamento da exceção apropriada. No teste de remoção é verificado se o objeto é realmente removido.

3.2. Serviço de Gerenciamento de Replicação - SGR

O conjunto de testes para o Serviço de Gerenciamento de Replicação consiste dos testes do Gerenciamento de Propriedades e do Gerenciamento de Grupos.

A cada grupo de objetos é associado um conjunto de propriedades e o SGR possibilita que estas sejam definidas por padrão ou pelo tipo do objeto, no momento da criação do grupo ou dinamicamente (com restrições). Os testes do Gerenciamento de Propriedades verificam o funcionamento dos métodos para manipulação dos conjuntos de propriedades, determinando se propriedades definidas por padrão ou por tipo são realmente atribuídas e se as devidas exceções são lançadas caso propriedades não suportadas ou com valores incorretos sejam passadas ao gerenciador. Certas propriedades como o estilo de replicação e o número inicial de réplicas não podem ser definidas dinamicamente.

O controle de filiação dos grupos de objetos pode ser feito pela infra-estrutura ou pela aplicação. No primeiro caso, o gerenciador de replicação invoca fábricas, nas localizações apropriadas, para criar os membros do grupo tanto inicialmente, para satisfazer a propriedade "número inicial de réplicas", quanto depois da perda de um membro devido a uma falha detectada, visando satisfazer a propriedade "número mínimo de réplicas". No segundo caso, a interface de gerenciamento de grupos do SGR permite que a aplicação crie um membro de um grupo, adicione um membro já existente ou remova, citando a localização do membro criado, adicionado ou removido. Desta forma, foram criados dois testes distintos que juntos verificam exaustivamente o funcionamento correto dos métodos da interface de Gerenciamento de Grupos.

3.3. Serviço de Gerenciamento de Falhas - SGF

A especificação FT-CORBA define interfaces para a comunicação dos detectores com os objetos monitorados e uma interface para a comunicação com o notificador. A especificação, no entanto, não define a quantidade e o arranjo dos detectores, os filtros do notificador, nem o funcionamento dos analisadores. Os testes do SGF não criam detectores específicos, e sim testam o funcionamento correto dos detectores implementados no GROUPPAC, que são detectores *pull* em 2 níveis (*host* e objeto) e um notificador com filtro simples que elimina relatórios duplicados [Lung et al., 2001]. Cada *host* possui um detector em nível de objeto, detectando falhas individuais de objetos. Todos os membros do grupo de detectores em nível de *host* detectam falhas de cada um dos detectores em nível de objeto, além de detectarem falhas de membros do seu próprio grupo.

Sendo assim, os testes do SGF registram-se no notificador e injetam falhas de parada em *hosts* e objetos para verificar se os detectores enviam, após determinado tempo, os relatórios no formato e com as informações corretas, como o tipo de falha (*host* ou objeto) e sua origem. Quando uma falha de *host* é detectada, também é verificado se o detector em nível de *host* é retirado de seu grupo, o que é responsabilidade do SGR. Os testes também consideram o fato de que não devem ser enviados relatórios de falhas de objetos de um grupo que tem a propriedade de estilo de monitoramento "não monitorado".

3.4. Serviço de Gerenciamento de Recuperação e *Logging* - SRL

O SRL não tem interfaces definidas pois a aplicação nunca interage diretamente com ele. As suas responsabilidades de recuperação de réplicas faltosas a partir de estados armazenados e requisições interceptadas (*logging*) dependem do estilo de replicação usado.

No estilo de replicação passiva, o SRL deve substituir um membro primário falho por um membro reserva correto. Para verificar se isto ocorre corretamente, os testes injetam falhas

nas réplicas primárias e verificam se os estados das réplicas reservas (substitutas) são consistentes com os estados das réplicas primárias faltosas no momento em que as reservas assumem o lugar das faltosas. Outra funcionalidade testada é envio de *checkpoints* periódicos as réplicas.

Em um grupo com replicação ativa, somente é necessário testar a atualização feita pelo SRL do estado de uma réplica que acaba que se juntar ao grupo, uma vez que as características deste tipo de replicação asseguram a consistência do estado das réplicas.

3.5. Considerações sobre os Testes

O GROUPPAC é um sistema relativamente grande, com aproximadamente **39**.**000** linhas de código (Java e IDL) distribuídas por mais de **120** classes (sem considerar os arquivos gerados pela tradução de interfaces em IDL e os testes). Levando este fato em consideração, os casos de testes utilizados se baseiam em grande parte nos comportamentos especificados para os serviços definidos no documento do FT-CORBA (cap. 23 de [Object Management Group, 2002]).

A ferramenta ANT [Apache Software Foundation, 2005] foi utilizada para automatizar o processo de testes em conjunto com suas extensões para a ferramenta de testes JUNIT [JUnit, 2005] e a ferramenta de avaliação de cobertura de testes COBERTURA [Doliner and Thomerson, 2005].

Ao todo são **19** testes que alcançaram uma taxa de cobertura de código de **78**%. É um bom resultado, haja vista a abordagem simplificada utilizada e a existência de trechos de código descontinuados (e não mais utilizados) do GROUPPAC (como as extensões de larga escala).

De fato, a aplicação dos testes mostrou-se efetiva no sentido que diversos erros foram detectados no código, e posteriormente corrigidos. Uma grande vantagem da abordagem usada é a capacidade de executar os testes sempre que alterações forem feitas no sistema, verificando desta forma sua correção em relação aos casos de uso definidos. E já que os testes atuam como um usuário da infra-estrutura, acessando apenas as interfaces do FT-CORBA, podem ser aplicados em qualquer outra infra-estrutura que siga a mesma especificação.

4. Considerações Finais

A aplicação de testes em *middleware* TF é uma atividade essencial para garantir a confiabilidade deste importante componente das aplicações críticas distribuídas. Este artigo relata a estratégia de testes utilizada no desenvolvimento do sistema GROUPPAC. A estratégia implementada é pragmática e se baseia na automação dos testes através de ferramentas largamente utilizadas pela comunidade Java. Esta estratégia esta longe da perfeição, porém foi eficiente no sentido de que permitiu uma boa cobertura do código do sistema e identificou vários erros antes não percebidos.

Referências

Apache Software Foundation (2005). Ant homepage. `http://ant.apache.org`.

Beck, K. and Gamma, E. (1998). Test-infected: Programmers love writing tests. *Java Report*, 3(7).

Bessani, A. N., Alchieri, E. A. P., Lung, L. C., and da Silva Fraga, J. (2004). GroupPac 3: Estendendo o FT-CORBA para Gerenciamento e Replicação Ativa. In *Anais do WTF 2004*, Gramado, RS.

Doliner, M. and Thomerson, J. (2005). Cobertura homepage. `http://cobertura.sourceforge.net`.

JUnit (2005). JUnit homepage. `http://www.junit.org`.

Kaner, C., Falk, J., and Nguyen, H. Q. (1999). *Testing Computer Software*. Willey, 2nd edition.

Lung, L. C., da Silva Fraga, J., Padilha, R., and Souza, L. (2001). Adaptando as especificações FT-CORBA para redes de larga escala. In *Anais do SBRC 2001*, Florianópolis, SC.

Object Management Group (2002). The Common Object Request Broker Architecture: Core Specification v3.0. OMG Standart formal/02-12-06.

Comunicação Fim-a-Fim a Alta Velocidade em Redes Gigabit

**Danilo M. Taveira, Igor M. Moraes, Daniel de O. Cunha,
Rafael P. Laufer, Marco D. D. Bicudo, Miguel E. M. Campista,
Aurelio Amodei Junior e Otto Carlos M. B. Duarte***

[1]Grupo de Teleinformática e Automação
PEE/COPPE-DEL/POLI
Universidade Federal do Rio de Janeiro

Resumo. *Com o aumento na largura de banda disponível e a possibilidade de transmissão de dados a Gigabits por segundo, os discos rígidos passam ser o principal gargalo em comunicações a altas taxas. Como solução, é mostrado que a utilização de diversos discos em paralelo possibilita taxas compatíveis com as exigências atuais das redes de comunicação.*

1. Introdução

Os enormes avanços tecnológicos obtidos na área de integração de circuitos permitiram ganhos extraordinários na capacidade de processamento dos computadores e na velocidade das redes de computadores. Embora uma rede com capacidade para transmitir a Gigabits por segundo permita um caminho de alta velocidade entre dois pontos, não é trivial o uso eficaz de toda esta taxa de transferência por uma aplicação. Alguns cuidados na escolha dos equipamentos de extremidade e na configuração dos protocolos a serem usados são essenciais para atingir as altas taxas de transmissão oferecidas.

Atualmente, um computador pessoal possui um alto poder de processamento com relógios da ordem de gigahertz. O barramento de memória, responsável pela comunicação direta entre o processador e a memória principal já possui relógios com centenas de megahertz. Desta forma, este barramento não representa mais um gargalo significativo no serviço a altas taxas. Os discos, entretanto, continuam sendo dispositivos com partes mecânicas, incapazes de acompanhar a velocidade dos componentes puramente eletrônicos.

A tecnologia RAID (*Redundant Array of Inexpensive Disks*) foi proposta inicialmente por Patterson *et al.* em 1988 [Patterson et al., 1988].No RAID nível 0, os dados são distribuídos uniformemente entre dois ou mais discos. Esse nível de RAID aumenta a taxa agregada obtida ao permitir que requisições de leitura e escrita aos discos sejam atendidas quase que simultaneamente.Entretanto, no RAID nível 0 nenhuma redundância é empregada.

O artigo está organizado da seguinte forma. A Seção 2 descreve os procedimentos adotados para a realização dos testes e os equipamentos envolvidos. Detalhes referentes à análise dos resultados são discutidos na Seção 3. Por fim, na Seção 4 são apresentadas as conclusões.

2. O Ambiente de Testes

Para avaliar o desempenho dos usuários finais em uma comunicação a altas taxas, desenvolveu-se um ambiente de testes. O ambiente é formado por dois computadores pessoais providos de interfaces de rede Gigabit Ethernet e interconectados através de um

*Este trabalho foi realizado com recursos do CNPq, CAPES, FAPERJ, FINEP, RNP e FUNTTEL.

comutador. Mais detalhes sobre a especificação de cada um dos equipamentos usados são apresentados na Tabela 1. A configuração do ambiente e dos equipamentos foi concebida para permitir a realização de testes com componentes da arquitetura de um computador pessoal, como as interfaces de rede e os discos rígidos, e também com os protocolos usados na comunicação.

Tabela 1: Especificação dos equipamentos usados no ambiente de testes.

Equipamento	**Especificação**
Computadores A e B	Processador Intel Pentium IV com relógio de 3,2 MHz
	Placa mãe Intel SE7210TP1-E com interface de rede Gigabit Ethernet integrada
	Memória RAM Kingston DDR 400 MHz Dual-Channel, 1 GB
	Disco rígido Seagate SCSI-320 10000 rpm (36GB)
	Sistema operacional Debian Linux 3.1 com núcleo 2.4.20
Comutador	Comutador D-Link DGS-3308TG com 6 portas Gigabit Ethernet

Os testes realizados envolvem a verificação experimental da vazão máxima obtida em uma rede Gigabit e a determinação do gargalo da comunicação a altas taxas. Em um primeiro cenário, busca-se determinar a taxa máxima de transmissão de dados entre os dois computadores. Os testes usam o protocolo de transporte UDP (*User Datagram Protocol*). O tráfego é gerado com a ferramenta `Iperf` [Iperf, 2005].

Além disso, busca-se analisar situações onde o objetivo é servir conteúdo armazenado em disco a taxa de um Gigabit por segundo. Nestes experimentos, um computador é usado como servidor de conteúdo a altas taxas. A especificação deste computador é a mesma apresentada na Tabela 1. O objetivo dos testes é comprovar que um computador provido de apenas um disco rígido não consegue transferir dados a um Gigabit por segundo. A partir daí, busca-se mostrar que o RAID nível 0 é uma alternativa para garantir a transferência de dados a altas taxas.

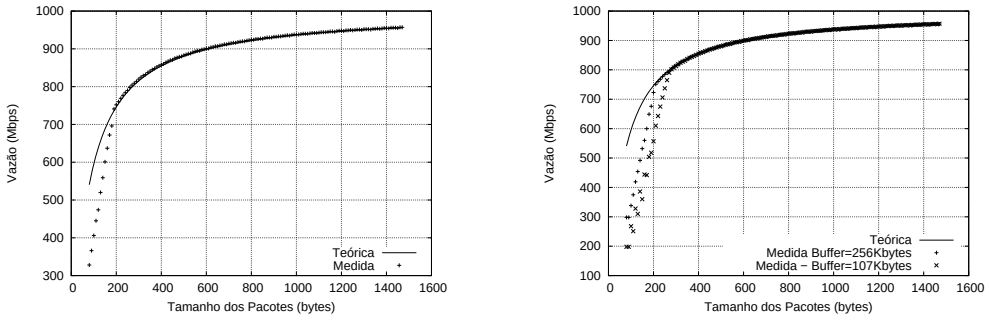
Para determinar a capacidade de transferência dos discos rígidos são consideradas três situações. Na primeira, o computador possui apenas um disco. Na segunda situação, considera-se que o computador está equipado com dois discos operando em RAID, nível 0. Por fim, na terceira o computador trabalha com três discos em RAID0. A ferramenta `IOMeter` [IOMeter, 2005] é usada nos testes com os discos rígidos

Nos testes também são analisados dois modos de implementação do RAID nível 0. O RAID implementado por software e o RAID híbrido. A implementação por software é em geral mais flexível do que a implementação que utiliza placas controladoras dedicadas. No entanto, o preço pago pela flexibilidade é o maior consumo de processamento, já que mais operações precisam ser feitas a cada leitura ou gravação. O suporte ao RAID é nativo no núcleo 2.4.20 do Linux. Entretanto, é necessário usar um conjunto de ferramentas, chamado de `Raidtools`, para criar a matriz de discos. A implementação híbrida está disponível na placa controladora RAID integrada à placa mãe do computador usado nos testes. No RAID híbrido, a controladora só implementa por hardware as funções básicas de acesso ao disco que são usadas pelas interrupções da BIOS. Estas interrupções permitem que todos os sistemas operacionais acessem o disco de uma forma padronizada. Só após acessarem os discos, os sistemas operacionais acessam diretamente a controladora RAID. Além disso, após o sistema operacional carregar o *driver* da controladora, todas as funções de RAID são implementadas por software. Dessa forma, a implementação híbrida pode ter um desempenho similar à implementação por software, ou até mesmo pior, caso o *driver* não seja otimizado.

3. Resultados

Inicialmente mediu-se a taxa máxima de transferência de dados obtida com o uso da rede Gigabit. O protocolo de transporte UDP é usado neste teste e os dados a serem transmitidos são lidos diretamente da memória RAM.

A Figura 1(a) mostra a taxa de dados obtida em função do tamanho do pacote enviado. Comparando a curva experimental com a curva teórica, nota-se que o computador consegue transmitir os dados em uma taxa próxima da taxa máxima teórica, a partir de um dado tamanho de pacote. Para pacotes abaixo de 200 bytes, é possível notar que a diferença é bastante expressiva. Esta queda ocorre por que é necessário um tempo mínimo para que cada pacote possa ser gerado.



(a) Taxa máxima de transmissão. (b) Taxa máxima de recepção.

Figura 1: Taxa de transferência da rede em função do tamanho do pacote.

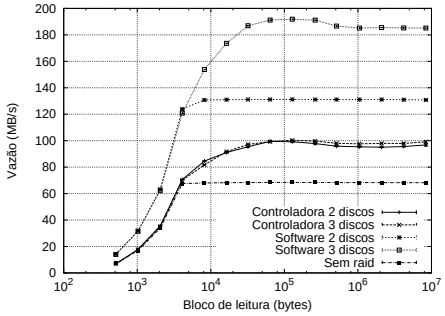
Pode-se observar na figura 1(b) que a taxa de dados recebida é menor que a taxa de dados transmitida. Esse fato indica que o tempo necessário na recepção de um pacote é superior ao tempo necessário para a sua geração. Isto se comprova com uma análise mais atenta da Figura 1(b), onde se pode perceber que o aumento do *buffer* do receptor diminui este efeito. Os resultados demonstram, assim, que o processamento e o barramento não são gargalos significativos na comunicação em alta velocidade, desde que os dados não sejam enviados em pacotes muito pequenos.

Após a analise da transmissão a altas taxas de dados lidos diretamente da memória, são realizados testes com o intuito de verificar o impacto do acesso ao disco no serviço de dados. O número de discos e o tipo de implementação do RAID nível 0 são variados. Também são avaliadas as implementações de RAID por software e híbrida, implementada pela controladora.

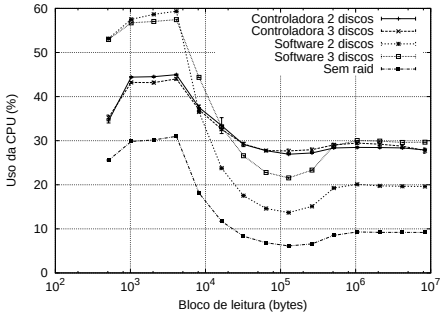
A Figura 2 mostra os resultados experimentais para a taxa de leitura fornecida pelos discos rígidos ao lerem diferentes tamanhos de blocos de dados seqüenciais. Dados seqüenciais são aqueles que estão armazenados de forma contígua no disco rígido.

Pode-se ver na Figura 2(a) que o RAID0 usando a controladora existente na placa mãe é menos eficiente para a leitura do que, o implementado por software. O melhor desempenho do RAID0 por software indica que os seus *drivers* são mais otimizados. De acordo com resultados obtidos, a vazão máxima dos discos só é alcançada quando são lidos blocos grandes de dados. Isso ocorre, pois para blocos pequenos, a probabilidade dos dados estarem em apenas um disco é grande, impossibilitando acessos simultâneos. Na Figura 2(b) é mostrado o uso da CPU durante o teste. Para blocos de leitura grandes, a diferença entre os resultados obtidos pelas duas implementações é significativamente menor.

Para analisar a velocidade de escrita em disco, são realizados testes semelhantes aos descritos anteriormente para a leitura. A Figura 3 mostra os resultados destes testes.

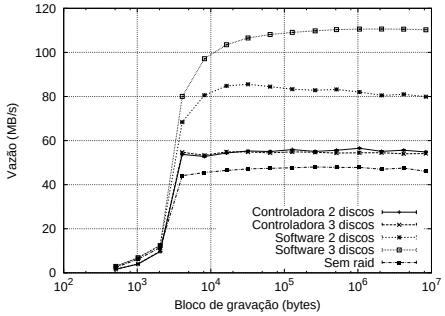


(a) Taxa de leitura.

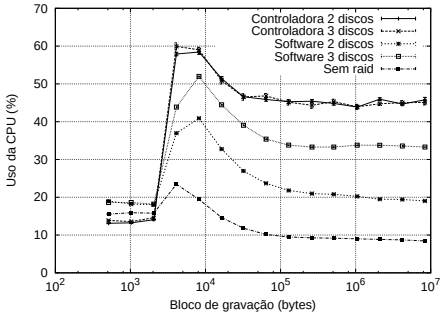


(b) Uso da CPU durante os testes.

Figura 2: Leitura de blocos seqüenciais.



(a) Taxa de escrita.



(b) Uso da CPU durante os testes.

Figura 3: Escrita de blocos seqüenciais.

No teste de escrita, é possível observar novamente que o desempenho do RAID0 implementado por software foi superior ao híbrido. A Figura 3(b) mostra o uso da CPU durante o teste de escrita seqüencial. Nessa situação, o uso da CPU pela implementação por software é menor se comparado ao uso da CPU usando-se o RAID0 da controladora. Os resultados mostram também que o custo da operação de escrita dos dados é maior que o de leitura, já que a capacidade máxima de escrita é menor do que a de leitura. Também é possível observar que mesmo com três discos em RAID0, não é possível obter uma taxa de 1 Gbps.

4. Conclusão

Este artigo analisou as principais limitações às comunicações em altas taxas impostas pela atual arquitetura dos computadores pessoais.

O resultados demonstram que o nó emissor é capaz de ocupar todo o meio de acordo com o tamanho dos pacotes utilizados. Foi também demonstrado que os discos rígidos constituem o principal gargalo nessas comunicações. Os testes demonstraram que o uso do RAID nível 0 consegue atender às exigências de comunicações a altas taxas e que a forma com que ele é implementado impacta significativamente os resultados.

Referências

IOMeter (2005). IOMeter. `http://www.iometer.org/`.

Iperf (2005). Iperf - The TCP/UDP Bandwidth Measurement Tool. `http://dast.nlanr.net/Projects/Iperf/`.

Patterson, D. A., Gibson, G. e Katz, R. H. (1988). A case for redundant arrays of inexpensive disks. Em *ACM SIGMOD'88 - International Conference on Management of Data*, páginas 109–116.

Fast Adaptable Uniform Consensus Using Global State Digests

Andrey Brito[1], **Francisco Brasileiro**[1,2], **Walfredo Cirne**[1,2]
Universidade Federal de Campina Grande
[1]Coordenação de Pós-Graduação em Informática
[2]Coordenação de Pós-Graduação em Engenharia Elétrica
Av. Aprígio Veloso, 882 - 58.109-970, Campina Grande, PB, Brazil
Phone: +55 83 310 1433 Fax: +55 83 310 1365
{fubica,andrey,walfredo}@dsc.ufcg.edu.br

Abstract

Protocols that solve the consensus problem have been widely recognized as important building blocks for the design of reliable distributed systems. This fact explains why considerable amount of work has been devoted both to establish the minimal system requirements that allow a solution to the problem, as well as to provide efficient protocols to solve it. We propose the use of global state digests *to design efficient and adaptable consensus protocols. A global state digest is a bounded and consistent summarized representation of the local states of all processes that run the protocol. By frequently providing processes with new global state digests, it is possible to allow processes to terminate the protocol soon after the minimal condition necessary to solve the problem holds, whatever are the contention levels experienced by the system. We present the design of a family of fast adaptable consensus protocols using this abstraction. Further, a global state digest provider can be implemented whenever the same assumptions required to implement perfect failure detectors hold (basically the ability to convey a bounded amount of information within a bounded interval of time).*

Keywords*: agreement protocols; perfect failure detection; consistent global state; synchronous and asynchronous distributed systems; wormholes.*

1 Introduction

In this paper we focus on the fundamental problem of reaching consensus among a set of n processes that communicate exclusively via the exchange of messages [11]. In the consensus problem, each process p_i proposes a value v_i and every correct process must decide for the same common value v among the values proposed, despite the possible crashes of up to f processes, $f < n$. Designing a protocol that guarantees these properties, however, is not trivial. There are two main sources of difficulties associ-

ated with the design of a fault-tolerant distributed consensus protocol, namely: i) the lack of synchrony guarantees provided by the underlying distributed system; and ii) the occurrence of failures in both processing and communication.

Synchronous systems provide strong synchrony guarantees for both processing schedule and communication delays. Because of this, detection of benign failures (such as crashes) within synchronous systems is straightforward. Therefore, solutions to the consensus problem for this kind of system have long been known [7]. On the downside, the strong synchrony guarantees provided by synchronous systems require an a priori knowledge of the worst case workloads to which the system will ever be subject. This makes it more difficult to develop such systems and reduces the applicability of fault-tolerant distributed protocols designed to execute over a synchronous system.

On the other hand, in a pure asynchronous system model, no synchrony guarantees are given (in fact the very notion of time is absent), thus, any distributed system can be considered a pure asynchronous system. Unfortunately, it is well known that most practical fault-tolerant distributed problems are impossible to be solved in this kind of system. Particularly, it has been proved that there is no deterministic solution to the consensus problem in pure asynchronous systems subject to even a single crash failure [12].

This result has prompted researchers to seek for the minimal synchrony guarantees that a distributed system must provide in order to allow fault-tolerant solutions to fundamental distributed problems, such as consensus. This effort is backed up by the observation that, although most off-the-shelf distributed systems are not synchronous, they do have some level of synchrony, and are, therefore, generally classified as partially synchronous systems [8, 10].

Within this context, the abstraction of unreliable failure detectors [5] is an important contribution. An unreliable failure detector is a "black-box" that is associated with each process of an asynchronous distributed system and encapsulates the synchrony required to solve fault-tolerant distributed problems. This synchrony is specified via a pair of properties that describe the behavior of classes of failure detectors. They define the failures that the failure detectors are able to detect (their *completeness* property) and the mistakes that they are allowed to make (their *accuracy* property). It has been shown that the weakest failure detector class that allows a deterministic solution to consensus, named $\diamond S$, must provide the following properties: eventually every process that crashes is permanently suspected by every correct process (*strong completeness*); and, there is a time after which some correct process is never suspected by any correct process (*eventual weak accuracy*) [6].

Following this result, a number of consensus protocols based on failure detectors of the class $\diamond S$ have been proposed (*e.g.* [16, 14, 13, 9, 15]). They all share the following characteristics: i) they are only able to tolerate a minority of faulty processes, *i.e.* they require $f < n/2$; and, ii) they are *indulgent* [9] in relation to the failure detector, *i.e.* even if the failure detector misbehaves and does not provide any synchrony guarantees, the safety properties of consensus are preserved. Other solutions to the consensus problem based on stronger failure detectors have also been proposed [5, 1]. They are not indulgent towards the failure detector, but allow greater resilience to faults (typically, they are able to tolerate up to $n - 1$ faults).

Implementations of stronger failure detectors require more synchrony guarantees from the underlying distributed system, which, as mentioned before, are more difficult to be found in most systems available

today. To overcome this problem, hybrid architectures (*wormholes*) may be used [18, 17]. The idea is to design protocols that assume an asynchronous system model and make no explicit requirement for synchrony guarantees, relying only on a strong failure detector (for instance, a perfect one) that encapsulates the required synchrony. In order to implement a perfect failure detector, the asynchronous system is augmented with a synchronous subsystem that is solely used to support the implementation of the failure detector. Note that the synchronous subsystem corresponds to just a small and well defined portion of the whole system, thus, rendering the difficult task of engineering such subsystem much simpler.

In a previous work [3] we discussed the implementation of a hybrid system, which relied on a hardware-based synchronous subsystem. The usage of this system was exemplified with a service, the Global State Digest Provider service (GSDP), that could be used to solve consensus. Later, we showed how a cheap and safe synchronous subsystem could be built based on COTS components [4]. In this paper, we give a detailed formalization of the GSDP abstraction. We then use the abstraction to design a consensus protocol, providing detailed description of the protocol as well as correctness proofs and a performance evaluation.

The rest of the paper is structured in the following way. Section 2 describes the system model assumed, gives a general description of the abstraction of a global state digest provider, and formally defines the uniform consensus problem. In Section 3, we present a family of uniform consensus protocols based on a particular instantiation of the global state digest provider abstraction (the proof of correctness of the protocols proposed is presented in Appendix A). In Section 4 we evaluate the performance of the proposed protocols and compare their performance with that of other protocols previously published. Section 5 concludes the paper with our final remarks.

2 System Model and Definitions

System Model The system model is patterned after the one described in [12]. It consists of a finite set Π of n processes, $n > 1$, namely, $\Pi = \{p_1, \ldots, p_n\}$. A process can fail by *crashing*, *i.e.* by prematurely halting, and a crashed process does not recover. A process behaves correctly (*i.e.* according to its specification) until it (possibly) crashes. At most f processes, $f < n$, may crash.

Processes are completely connected via a reliable network. Processes communicate with each other by message passing through the communication channels that comprise the reliable network: there are no message creation, alteration, duplication or loss. Further, there are assumptions neither on the relative speed of processes nor on message transfer delays.

The Global State Digest Provider To circumvent the impossibility result of [12] and allow the implementation of fast and adaptable consensus protocols, we consider that the system is augmented with a synchronous subsystem that implements the abstraction of a *global state digest provider* (GSDP). A *global state digest* (GSD) is a protocol-specific summarized description of the relevant events that happened within the system during a particular time interval, including an indication of the processes that have crashed.

Each process has access to the GSDP via its local GSDP module. This module is able to perform

some bounded computation in a timely way and to exchange, through a reliable network, a bounded number of messages whose sizes are also bounded. These restrictions allow end-to-end communication delays between any two correct GSDP modules to be bounded by a known constant δ. We assume that each GSDP module has access to a local clock that measures a clock interval time Δt in a real time interval $\Delta t'$, $\Delta t \times (1 - \rho) \leq \Delta t' \leq \Delta t \times (1 + \rho)$, where ρ is a known constant. Moreover, we assume that a process and its associated GSDP module form a single unit, thus a crashed (resp. correct) process also implies a crashed (resp. correct) GSDP module. This assumption is supported by the following arguments: (1) it is possible to implement the system such that both the GSDP and the process execute in the same node [4], thus, if the node crashes, the GSDP and the process will both fail; (2) if the GSDP could fail alone, the process remains stopped since it depends on the GSDP to make progress; also, it would not be considered in any distributed computation since the GSDP would not propagate its information; finally, (3) if the process fails the operating system will break the link between the process and the GSDP (for example, this link could be through an opened special file), and the GSDP will stop, propagating the failure information.

The Uniform Consensus Problem Before presenting protocols to solve the uniform consensus problem, let us recall the formal definition of the problem. In the uniform consensus problem, every process p_i *proposes* a value v_i and all correct processes have to *decide* on some value v, in relation to the set of proposed values. More formally, the uniform consensus problem is defined by the following properties [12]:

- **termination**: every correct process eventually decides some value;
- **uniform integrity**: every process decides at most once;
- **uniform validity**: if a process decides for the value v, then v was proposed by some process; and
- **uniform agreement**: no two processes decide differently.

3 Designing Uniform Consensus Protocols Supported by a Global State Digest Provider

We start the presentation of the protocols by defining the data structure of the GSDs that are delivered by the GSDP. For the consensus[1] protocol presented in this paper, a suitable GSD contains the following fields:

- *detection_vector*: a status vector with n bits, in which element i represents the operational status of process p_i (it is initially set to 0, and it is set to 1 if p_i is faulty);
- *reception_matrix*: a $n \times n$ matrix of bits in which the bit $[i, j]$ indicates whether p_i has received a message from p_j or not (it is initially set to 0, and it is set to 1 if the message has been received); and
- *consensual_identity*: an identity field with $\lceil log_2 n \rceil$ bits used to hold the identity of the process that provides the consensual value for the protocol (it is initially set to $\perp$).

[1] For the sake of brevity, from now on we will use simply the term consensus when referring to uniform consensus.

The family of consensus protocols supported by a GSDP have a synchronous part and an asynchronous part. There are two parameters that differentiate each member of the family of protocols. The first one, named *quorum*, defines the maximum number of processes that are necessary to "elect" the process that provides the consensual value. This parameter affects only the synchronous part of the protocol. The value of *quorum* is such that $f + 1 \leq quorum \leq n$. The second parameter, named *proposers*, defines the number of processes that are allowed to propose a value in the execution of the protocol. It affects only the asynchronous part of the protocol, and its value is such that $f + 1 \leq proposers \leq n$. A pair of values for the parameters *quorum* and *proposers* defines a particular protocol. Thus, each member of the family of protocols is denoted by *GSDP-consensus(q,p)*, where q and p are the *quorum* and *proposers* parameters, respectively. In the next two subsections we describe the functioning of the synchronous and asynchronous parts of the protocols.

3.1 The Synchronous Part of the Protocol GSDP-consensus(q, p)

The synchronous part of the protocol is executed by the GSDP modules that are associated with each process executing the consensus protocol. They exchange digests with the relevant events locally perceived in order to form *consistent* GSDs. More formally, a GSDP provides the following properties:

- **strong completeness of detection**: if some process p_j crashes, then eventually all GSDs delivered by every GSDP module have $detection\_vector[j] = 1$;
- **strong accuracy of detection**: if any GSD delivered has $detection\_vector[j] = 1$, then p_j has indeed crashed;
- **strong completeness of reception**: if some correct process p_i receives a message from some process p_j, then eventually all GSDs delivered by every GSDP module have $reception\_matrix[i, j] = 1$;
- **strong accuracy of reception**: if any GSD delivered has $reception\_matrix[i, j] = 1$, then p_i has indeed received a message from p_j;
- **validity**: if a GSD g has $g.consensual\_identity = x$ and f_{actual} is the number of entries in $g.detection\_vector$ set to 1 (*i.e.* the number of processes whose failures have been detected), then, there are at least $q - f_{actual}$ occurrences of i, $1 \leq i \leq n$, that satisfy: $g.detection\_vector[i] = 0 \wedge g.reception\_matrix[i, x] = 1$; also, if g has $g.consensual\_identity = \perp$, then, there is no p_x such that, there are at least $q - f_{actual}$ occurrences of i, $1 \leq i \leq n$, that satisfy: $g.detection\_vector[i] = 0 \wedge g.reception\_matrix[i, x] = 1$; and
- **write-once**: if a GSD is ever delivered with $consensual\_identity = x \neq \perp$, then every GSD delivered by any GSDP module that has $consensual\_identity \neq \perp$ also has $consensual\_identity = x$.

Each GSDP module $gsdp_i$ maintains two GSDs variables, named $ready\_for\_delivery_i$ and $local_i$. Initially, they have all bits of their $detection\_vector$ and $reception\_matrix$ set to 0, and their $consensual\_identity$ fields are set to $\perp$. To access its local GSDP module, a process p_i invokes the *getGSD* primitive. When p_i invokes this primitive, it receives in return the contents of the $ready\_for\_delivery_i$ variable. The $local_i$ variable is used to locally compute valid GSDs. Whenever a valid GSD is formed by a GSDP module $gsdp_i$, it copies $local_i$ into $ready\_for\_delivery_i$, thus making this newly formed GSD available for delivery.

To guarantee the *strong completeness of detection* and the *strong accuracy of detection* properties, the GSDP modules implement a perfect failure detector over the synchronous subsystem. The simplest way to implement such a failure detector in a synchronous system is to have every module broadcasting empty *heartbeat* messages at some a priori agreed rate. Assume that every correct GSDP module broadcasts a heartbeat message every τ units of time. Let $d = (\tau + \delta)(1 + \rho)$ and t_0 be a time after which all processes have already started the execution of the protocol. Thus, the failure of any GSDP module that happens at some time t, $t \geq t_0$, is detected by every correct module at latest by $t + d$. If a GSDP module $gsdp_i$ detects the failure of some process p_j, then it updates its $local_i$ GSD, such that $local_i.detection\_vector[j] = 1$. This guarantees the *strong completeness of detection* property, provided that $local_i$ eventually gets valid and is, therefore, copied into $ready\_for\_delivery_i$. Further, the upper bound on end-to-end communication delays of the synchronous subsystem (δ), together with the bound on the maximum drift rate (ρ), guarantee that detection is perfect, hence the *strong accuracy of detection* property also holds.

The *strong completeness of reception* and the *strong accuracy of reception* properties are also trivially met. Whenever a process p_i receives a protocol message from a process p_j, its associated GSDP module $gsdp_i$ is notified. Then, $gsdp_i$ sets $local_i.reception\_matrix[i, j] = 1$ and broadcasts a $\lceil log_2 n \rceil$-bit long message containing the index j of the element that has been set on its $local_i.reception\_matrix$ to all other GSDP modules. When a GSDP module $gsdp_k$ receives such a message it sets the corresponding element of its $local_k.reception\_matrix$. This guarantees *strong accuracy of reception*, provided that the local GSDs eventually get valid and are, therefore, copied into the respective $ready\_for\_delivery$ GSDs. Further, since the messages broadcast by a correct GSDP module are always received by all correct GSDP modules, *strong completeness of reception* is also guaranteed.

The most challenging aspect of the implementation of the GSDP is to guarantee that the *validity* and *write-once* properties are simultaneously met. The *validity* property is easy to be met, since it depends only on the local information gathered by a GSDP module. The difficulty arises when, after updating either the $detection\_vector$ or the $reception\_matrix$ of its $local$ GSD, a GSDP module $gsdp_i$ realizes that it must update $local_i.consensual\_identity$ (otherwise, no more valid GSDs may be made available). To preserve *validity*, from this point onwards, $local_i$ can only be copied into $ready\_for\_delivery_i$ if $local_i.consensual\_identity \neq \perp$. However, since it is possible that a GSDP module $gsdp_j$ fails while sending its heartbeat messages or while sending information about its $reception\_matrix$, some GSDP modules may receive such messages, while others do not. Thus, different GSDP modules might have different views on the value that should be used to set the $consensual\_identity$ field. Therefore, to guarantee the *write-once* property, they must elect a common process. There are several possibilities to achieve this, the simplest one being for a GSDP to atomically broadcast a message with its local view of which process should be elected. The first message that is atomically delivered defines the identity of the process to be used by all GSDP modules. The atomic broadcast protocol guarantees that the messages that are delivered by some GSDP module are delivered by every correct GSDP module; further, any two messages that are delivered by some GSDP module are delivered in the same order by all GSDP modules that deliver them [7]. This guarantees that all modules update their local GSD with the same value. Once the value of the $consensual\_identity$ of every correct GSDP module is set to the same value x, $x \neq \perp$, this value will never change, thus guaranteeing that the *write-once* property of the GSDP is met.

Algorithm 1 The pseudo-code of $gsdp_i$ associated with process p_i that executes GSDP-consensus(q, p)

```
% shared variables and function
local_i.detection_vector = ready_for_delivery_i.detection_vector = 0 % set all bits to 0
local_i.reception_matrix = ready_for_delivery_i.reception_matrix = 0 % set all bits to 0
local_i.consensual_identification = ready_for_delivery_i.consensual_identification = ⊥

while true do
   sleep for τ units of time
   send gsdp_i's heartbeat to all GSDP modules
end while
||
while true do
   when p_j's failure has been detected
      local_i.detection_vector[j] = 1
      if local_i satisfies the validity property then ready_for_delivery_i = local_i end if
   end when
end while
||
while true do
   when p_i notifies that p_j's consensus protocol message has been received
      local_i.reception_matrix[i, j] = 1
      send index j to all GSDP modules
      if local_i satisfies the validity property then ready_for_delivery_i = local_i end if
   end when
end while
||
while true do
   when index k has been received from gsdp_j
      local_i.reception_matrix[j, k] = 1
      if local_i satisfies the validity property then ready_for_delivery_i = local_i end if
   end when
end while
||
decision_reached = false
while not decision_reached do
   when local_i does not satisfy the validity property
      atomically broadcast the identity x of a process p_x whose message has been received by all correct processes
      waituntil atomically deliver some identity y
      local_i.consensual_identity = y; ready_for_delivery_i = local_i; decision_reached = true
   end when
end while
```

Remark. It has been shown in [5] that a solution to atomic broadcast is also a solution to consensus. Thus, a natural question is: why not use this atomic broadcast service to solve consensus for the application? We recall that the bandwidth of the synchronous subsystem must be judiciously used, and since the upper bound on the size of the messages that the applications will ever generate is unknown, they cannot directly use the synchronous subsystem.

Algorithm 1 gives the pseudo-code of the concurrent tasks that implement the GSDP module associated with a process p_i that executes GSDP-consensus(q, p).

3.2 The Asynchronous Part of the Protocol GSDP-consensus(q, p)

The asynchronous part of the protocol is structured as three concurrent tasks. In the first task (the *proposition* task) p processes send messages to the other processes containing their proposition values. The second task (the *listening* task) is responsible for receiving and storing the proposition messages that have been sent by the other processes. It also notifies its associated GSDP module of the proposition

messages that have been received. The final task (the *decision* task) is responsible for detecting that a decision can be made and that the execution of the protocol can be terminated. The decision task is also very simple. It enters a loop constantly querying its local GSDP module. Whenever a GSD is delivered such that the *consensual_identity* field is equal to some x, $x \neq \perp$, it verifies if it has already received p_x's message. If not, it waits until p_x's message is received. In either case, after successfully retrieving p_x's message, it decides for the value contained in the message and terminates its execution of the protocol by forwarding p_x's message to every correct process that have not yet received it. Algorithm 2 is the pseudo-code of the concurrent tasks that implement the asynchronous part of the protocol.

Algorithm 2 The pseudo-code of GSDP-consensus(q, p) protocol executed by process p_i

```
% shared variables
bagOfMessages_i = ∅
decided_i = false

% Proposition task
when propose(v_i) is invoked
  if i ≤ p then send m_i(v_i) to all processes end if
end when
||
% Listening task
while not decided_i do
  when receive m_j(v_j) from p_j
    if m_j(v_j) not in bagOfMessages_i then
      add m_j(v_j) to bagOfMessages_i
      notify gsdp_i of the reception of a proposition message from p_j
    end if
  end when
end while
||
% Decision task
while not decided_i do
  g = getGSD()
  x = g.consensusal_identity
  if x ≠⊥ then
    waituntil m_x(v_x) in bagOfMessages_i
    m_x(v_x) = getConsensualMessage(x, bagOfMessages_i) % retrieves p_x's message
    send m_x(v_x) to all p_k such that g.detection_vector[k] = 0 ∧ g.reception_matrix[k, x] = 0
    decided_i = true
    return(v_x) % decides for the value proposed by p_x
  end if
end while
```

4 Performance Evaluation

In this section we analyze the performance of the protocols presented in this paper and compare their performance with that of other protocols previously published. Performance prediction of most consensus protocols for asynchronous systems augmented with unreliable failure detectors use a round-based model. Two metrics are normally used: i) the time complexity of the protocol, given by the number of communication rounds that the protocol needs to execute in a particular run; and ii) the message complexity, given by the number of messages that are sent by the processes executing a particular run of the protocol. In the following we compare the performance of several consensus protocols, including the GSDP-consensus(q, p) protocols, using these metrics.

In the GSDP-consensus(q, p) protocols, processes may send messages in two parts of the protocol:

when proposing their values, and, possibly, when diffusing the consensual message, just before deciding. Thus, considering the number of rounds, the protocols GSDP-consensus(q, p) require either 1 or 2 communication rounds, in the worst case. In particular, the protocols GSDP-consensus(n, p) require just a single communication round, and are, therefore, optimal with respect to this metric. It is worth noting that, like other protocols based upon strong failure detectors, the number of rounds is independent of the number of failures that actually occur during a particular run of the protocol.

The message complexity of the protocols also vary from protocol to protocol. In failure-free runs, the protocols GSDP-consensus(q, p), send $n.p$ messages in the proposition phase, while in the decision phase up to $n - q$ messages are sent by each process that decides. Thus, in failure-free runs, the message complexity of these protocols is: $n.p + n(n - q) = n(n + p - q)$. In summary, considering message complexity, the most expensive protocol is GSDP-consensus($f + 1$, n) with a cost of $n(2n - f - 1)$, while the cheapest protocol is GSDP-consensus(n, $f + 1$) with a cost of $n(f + 1)$. Finally, the message complexity of all protocols is reduced as failures occur (this is obvious, since faulty processes do not send messages).

Dutta and Guerraoui compare several indulgent consensus protocols considering a failure-free scenario and several scenarios where failures occur [9]. The consensus protocols that they propose in [9] achieve consensus in 2 rounds in all scenarios and are the most efficient among the protocols surveyed. It is important to point out that these results consider only runs in which the failure detector makes no mistake. For those runs in which the failure detector may make mistakes, the maximum number of rounds cannot be easily computed a priori. Considering message complexity, all indulgent consensus protocols require the execution of at least one reliable broadcast. Thus, the message complexity is always greater than $(n - 1)^2$.

Brasileiro *et al.* present a consensus protocol that is able to decide in a single round in "good" runs (those in which enough processes propose the same value), however, in the other runs the protocol relies on an underlying consensus protocol, therefore requiring more rounds to decide [2]. The first part of the protocol (that tries to achieve fast decision) requires n^2 messages, which is the message complexity of the protocol in "good" runs.

Protocols based on stronger failure detectors may require as much as n rounds to reach a decision. For instance, the consensus protocol that uses a failure detector of the class S presented in [5], always requires n rounds, each round requiring n^2 messages. Thus, in failure-free runs, the protocol requires n^3 messages to be sent.

5 Conclusion

In this paper we have detailed the global state digest provider (GSDP), an abstraction to support the implementation of distributed protocols. Informally, a *global state digest* is a bounded and consistent summarized representation of the local states of all processes that run the protocol (including their operational status: correct or crashed). It is a stronger abstraction than a perfect failure detector and, therefore, provides a more powerful framework for the design of fault-tolerant distributed protocols. Regarding the consensus problem, the main contribution of the GSDP is that it provides information that allows a consensus protocol to adapt itself to the fluctuations on the contention levels experienced by the system during a particular execution of the protocol - a feature that is present in no other consensus protocol

based upon perfect failure detectors. Nevertheless, the implementation of a GSDP requires no other assumptions than those already required to implement perfect failure detection, namely the existence of an (ideally fast) synchronous subsystem that is able to guarantee a known upper bound to the end-to-end communication delays between any two correct nodes.

Whether suitable GSDPs may be designed to support the design and implementation of other distributed protocols is an open issue that we are currently investigating. Nevertheless, we believe that a GSDP is a useful abstraction to any distributed problem that can take advantage of receiving GSDs with the following *monotonicity* property: let GSD_r be the set of all GSDs that happen in any run r of a protocol P that solves a given distributed problem, and assume that all elements in every GSD_r are totally ordered, such that for any two elements g and g' of GSD_r, either $g \rightarrow g'$ or $g' \rightarrow g$; let $\mathcal{P}$ be the set of predicates that P applies on the GSDs it is delivered on its run r, and g_1 and g_2 two elements of GSD_r, such that $g_1 \rightarrow g_2$; GSD_r is monotonic in relation to P iff $\forall p \in \mathcal{P}$, if p holds in g_1 then it also holds in g_2. In a recent paper, we have shown how a cheap and safe GSDP service could be built using off-the-shelf components such as Linux boxes connected via a switched Fast-Ethernet links [4].

References

[1] AGUILERA, M. K., LANN, G. L., AND TOUEG, S. On the impact of fast failure detectors on real-time fault-tolerant systems. In 16^{th} *International Symposium on Distributed Computing (DISC 2002)* (Tolouse, France, October 2002), pp. 354–369.

[2] BRASILEIRO, F. V., GREVE, F., MOSTEFAOUI, A., AND RAYNAL, M. Consensus in one communication step is possible. Research Report 1321, INRIA, 2001.

[3] BRITO, A., AND BRASILEIRO, F. Programando um subsistema síncrono para suporte a mecanismos eficientes de tolerância a falhas. In *Workshop de Tolerância a Falhas* (2004).

[4] BRITO, A., AND BRASILEIRO, F. A cheap and safe cots wormhole for local area network. In *Proceedings of the 10th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems (DPDNS'05)* (2005).

[5] CHANDRA, T., AND TOUEG, S. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM 43*, 2 (Mar 1996), 225–267.

[6] CHANDRA, T. D., HADZILACOS, V., AND TOUEG, S. The weakest failure detector for solving consensus. *Journal of the ACM 43*, 4 (Jul 1996), 685–722.

[7] CRISTIAN, F., AGHILI, H., STRONG, R., AND DOLEV, D. Atomic broadcast: from simple message diffusion to Byzantine agreement. In *Proceedings of the* 15^{th} *IEEE International Symposium on Fault-Tolerant Computing (FTCS'85)* (Ann Arbor, Jun 1985), pp. 200–206.

[8] DOLEV, D., DWORK, C., AND STOCKMEYER, L. On the minimal synchronism needed for distributed consensus. *Journal of the ACM 34*, 1 (Jan 1987), 77–97.

[9] DUTTA, P., AND GUERRAOUI, R. Fast indulgent consensus with zero degradation. In *Proceedings of the* $4th$ *European Dependable Computing Conference (EDCC4)* (Toulouse, France, Oct 2002).

[10] DWORK, C., LYNCH, N. A., AND STOCKMEYER, L. Consensus in the presence of partial synchrony. *Journal of the ACM 35*, 2 (Apr 1988), 288–323.

[11] FISCHER, M. J. The consensus problem in unreliable distributed systems. Research Report 273, Yale University, Jun 1983.

[12] FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. D. Impossibility of distributed consensus with one faulty process. *Journal of ACM 32*, 2 (Apr 1985), 374–382.

[13] HURFIN, M., AND RAYNAL, M. A simple and fast asynchronous consensus protocol based on a weak failure detector. *Distributed Computing 12*, 4 (1999), 209–223.

[14] MOSTEFAOUI, A., AND RAYNAL, M. Solving consensus using chandra toueg's unreliable failure detectors: a general quorum based approach. In *Proceedings of the* 13^{th} *International Symposium on Distributed Computing (DISC'99)* (Bratislava, Slovaquia, Sep 1999), pp. 49–63.

[15] SAMPAIO, L. M. R., BRASILEIRO, F. V., DA C. CIRNE, W., AND DE FIGUEIREDO, J. C. A. How bad are wrong suspicions? towards adaptive distributed protocols. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'2003)* (San Franciso, California, USA, June 2003), pp. 551–560.

[16] SCHIPER, A. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing 10*, 3 (Apr. 1997), 149–157.

[17] VERÍSSIMO, P. Uncertainty and predictability: Can they be reconciled? *Future Directions in Distributed Computing, Springer Verlag LNCS 2584* (May 2003), 108–113.

[18] VERÍSSIMO, P., AND CASIMIRO, A. The Timely Computing Base model and architecture. *Transactions on Computers - Special Issue on Asynchronous Real-Time Systems 51*, 8 (Aug. 2002).

Appendix A. Proof of Correctness of the GSDP-consensus(q, p) Protocols

Lemma 1 Every correct process that executes the protocol presented in Algorithm 2 eventually decides some value (*termination*).

Proof If a correct process ever decides it does so while executing the decision task of Algorithm 2. In this task a decision is made only if the process is delivered a GSD g, such that $g.consensual\_identity \neq \perp$. Therefore, to prove the lemma, we must first show that eventually every correct process is delivered such a GSD. From the *validity* property, a GSD g with $g.consensual\_identity \neq \perp$ can only be delivered if there is a p_x for which there are at least $q - f_{actual}$ occurrences of i, $1 \leq i \leq n$, such that $g.detection\_vector[i] = 0$ and $g.reception\_matrix[i, x] = 1$.[2] Since channels are reliable and at least p, $p \geq f + 1$, processes broadcasts their values to all processes, the proposition message of at least 1 process will be received by all processes that have not crashed, *i.e.* $n - f_{actual}$ processes. Without loss of generality, let p_x be such a process. The *strong completeness of reception* property guarantees that eventually, all GSDs delivered by every correct GSDP module have $reception\_matrix[i, x] = 1$, for every correct p_i. On the other hand, the *strong completeness of detection* property guarantees that eventually, all GSDs delivered by every correct GSDP module have $detection\_vector[j] = 1$ for every p_j that fails. Thus, eventually all GSDs are delivered such that there are $n - f_{actual}$ occurrences of i, $1 \leq i \leq n$, for which $detection\_vector[i] = 0$ and $reception\_matrix[i, x] = 1$. Since $n - f_{actual} \geq q - f_{actual}$, then one must have $decision\_identity \neq \perp$. Since the decision task keeps constantly querying the GSDP, eventually a GSD g with $g.consensual\_identity \neq \perp$ is delivered to every correct process.

To complete the proof we must show that the proposition message sent by the process whose identification is
$g.consensual\_identity$ has indeed been received by every correct process p_i, and therefore, can be retrieved from $bagOfMessages_i$. The *strong accuracy of detection* property guarantees that, for any correct process p_i, no GSDP module will ever deliver a GSD with $g.detection\_vector[i] = 1$, thus

[2] Notice that this condition does not guarantee that $g.consensual\_identity = x$, but only that $g.consensual\_identity \neq \perp$.

the correct processes are a subset of the processes that g indicates not to have failed (*i.e.* every p_i such that $g.detection\_vector[i] = 0$). The *strong accuracy of reception* property guarantees that if $g.reception\_matrix[i, x] = 1$ then p_i has indeed received p_x's message. Since $q \geq f+1$ and $f \geq f_{actual}$, then, there is at least 1 correct process among the $q - f_{actual}$ processes that have received p_x's message. This process will complete p_x's broadcast, thus even if p_x fails while broadcasting its message, it is guaranteed that every correct process will receive p_x's message. Hence the lemma. □

Lemma 2 Every process that executes the protocol presented in Algorithm 2 decides at most once (*uniform integrity*).

Proof This is trivially met by the protocol presented in Algorithm 2. As can be seen in the pseudo-code of the protocol, there is only one decision point for any process p_i that decides. Further, after deciding, a process terminates its execution of the protocol. □

Lemma 3 If a process that executes the protocol presented in Algorithm 2 decides for the value v, then v was proposed by some process (*uniform validity*).

Proof In Algorithm 2, the decision value of a process p_i is one that has been encapsulated in a message contained in $bagOfMessages_i$. The only messages that enter $bagOfMessages_i$ of a process p_i are the proposition messages sent by the processes executing the protocol. Hence the lemma. □

Lemma 4 No two processes that execute the protocol presented in Algorithm 2 decide differently (*uniform agreement*).

Proof Lemma 1 shows that every correct process decides. Let p_x be the process whose proposition is the decision value of some correct p_i. Thus, p_i must have been delivered a GSD g such that $g.consensual\_identity = x$. From the *write-once* property if any other process is delivered a GSD g' with $g'.consensual\_identity \neq \perp$ then it must have $g'.consensual\_identity = x$. Thus, any process that decides must also decide for the value proposed by p_x. Hence the lemma. □

Theorem 1 The protocol presented in Algorithm 2 solves consensus.

Proof The proof follows directly from lemmas 1, 2, 3 and 4. □

Integração das Especificações ROMIOP e ETF para Difusão Atômica no CORBA*

Daniel Borusch[1], Lau Cheuk Lung[1], Alysson Neves Bessani[2], Joni da Silva Fraga[2]

[1]PPGIA Programa de Pós-Graduação em Informática Aplicada
PUC-PR Pontifícia Universidade Católica do Paraná
R. Imaculada Conceição, 1155 Prado Velho CEP 80215-901 Curitiba PR

[2]DAS Departamento de Automação e Sistemas
UFSC Universidade Federal de Santa Catarina
Campus Universitário, Caixa Postal 476 CEP 88040-900 Florianópolis SC

{dborusch,lau}@ppgia.pucpr.br, {neves,fraga}@das.ufsc.br

Abstract. *OMG published a specification of a reliable ordered multicast inter-orb protocol for distributed applications developed in CORBA (ROMIOP). That specification was made to attend the demand of applications that need more restritive guarantees on reliability and ordenation, since already existed a specification without these resources (UMIOP). This article presents how ROMIOP was implemented as well as modifications that were made on the specification to make possible the creation of a better protocol. Perfomance measures were made comparing ROMIOP with others protocols, like UMIOP, to show it´s caracteristics and benefits against the rest.*

Resumo. *A OMG publicou uma especificação de um mecanismo de difusão confiável e ordenada para aplicações distribuídas desenvolvidas em CORBA (ROMIOP). Essa especificação foi criada para atender a demanda de aplicações que necessitam de garantias mais restritivas de acordo e ordenação, visto que já havia uma especificação sem esses recursos (UMIOP). Este artigo apresenta como foi implementado o ROMIOP bem como alterações que foram realizadas na especificação para possibilitar a criação de um protocolo melhor. Foram feitas medidas de desempenho comparando o ROMIOP com outros protocolos, como o UMIOP, para demonstrar as características e benefícios do mesmo em relação aos demais.*

1. Introdução

A arquitetura CORBA (*Common Object Request Broker Architecture*) [OMG, 2002a], criada pela OMG (*Object Management Group*), tem como principal componente o ORB (*Object Request Broker*). Ele possibilita que os objetos recebam e realizem invocações de modo transparente em um sistema distribuído, sendo considerado a base para a

* Este trabalho faz parte do projeto GroupPac/MJaco financiado pelo CNPq (Edital Software Livre 401802/2003-5).

interoperabilidade entre aplicações em ambientes heterogêneos. Para realizar a troca de mensagens entre os ORBs, há um elemento que especifica uma sintaxe de transferência padrão além de um conjunto de formatos de mensagens conhecido como GIOP (*General Inter-ORB Protocol*). A implementação do GIOP para o protocolo TCP/IP é conhecida como IIOP (*Internet Inter-ORB Protocol*), que usa como base à comunicação ponto a ponto, ideal para aplicações do tipo cliente/servidor. Entretanto, diversas outras áreas de aplicação necessitam disseminar a mesma mensagem para uma infinidade de *hosts*. Uma das formas de fazer isso é utilizando o *multicast* IP, que compreende um conjunto de extensões ao protocolo IP que o habilita na concretização de comunicações multiponto [Deering, 1986].

Visto que não havia nenhuma especificação que descrevia a utilização da comunicação multiponto na arquitetura CORBA, em 2001 a OMG publicou a especificação do UMIOP (*Unreliable Multicast Inter-ORB Protocol*) [OMG, 2003a]. O UMIOP foi proposto para prover um mecanismo comum de entrega de requisições sem garantias via *multicast*. O protocolo de transporte padrão definido para o UMIOP foi o *multicast* IP sobre UDP/IP, que diferentemente do TCP/IP, não é orientado à conexão, com isso apenas invocações *one-way* podem ser realizadas. Não tendo nenhuma garantia, o MIOP (protocolo definido na especificação UMIOP) pode ser caracterizado como sendo de alto desempenho, considerado ideal para aplicações como *streaming* de áudio e vídeo, em que a perda de alguns pacotes pode ser tolerada. Contudo, diversas aplicações não podem tolerar perdas de pacotes, necessitando de garantias mais restritivas como acordo e ordenação. Para isso, como a OMG não havia publicado até o momento nenhuma especificação que apresentasse uma solução, o nosso grupo de pesquisa propôs o protocolo ReMIOP (*Reliable Multicast Inter-ORB Protocol*) [Bessani et al., 2003] para atender a essa demanda.

Entretanto, no final de 2002, a OMG publicou uma especificação preliminar (*draft*), planejada para ser padronizada em 2005, que apresenta uma solução utilizando para isso um protocolo de comunicação multiponto com garantia de entrega e ordenação total (ou seja, difusão atômica [Défago et al., 2004]). Esta especificação foi denominada ROMIOP (*Reliable, Ordered, Multicast Inter-ORB Protocol*) [OMG, 2003b]. O ROMIOP, assim como o UMIOP, também utiliza o *multicast* IP sobre UDP/IP, porém invocações com retorno de resposta (*two-way*) são suportadas. Fazendo um estudo detalhado do ROMIOP é possível verificar que essa especificação fornece uma série de interfaces IDL, algumas ainda confusas, dando uma larga margem de interpretações. Ou seja, a especificação não dá muitos detalhes de como as interfaces devem ser implementadas e que algoritmos de ordenação total adotar [Défago et al., 2004].

Além dessas especificações, a OMG publicou recentemente a especificação ETF (*Extensible Transport Framework*) [OMG, 2003] que define um arcabouço (*framework*) que permite a terceiros projetar e implementar *plugins* adicionais de protocolos de transporte de mensagens GIOP no ORB. Esta especificação, que já é implementada na grande maioria dos ORBs disponíveis, torna possível a extensão de um ORB/CORBA através da adição de novos protocolos de transporte sem que haja modificações significantes na sua estrutura. No entanto, a ETF foi concebida visando apenas protocolos de transporte ponto-a-ponto, para multiponto, extensões na especificação são necessárias.

Este artigo propõe, como principal contribuição, um conjunto de extensões para efetivar a integração das especificações ROMIOP e ETF, englobando aspectos arquiteturais, conceituais e algumas decisões de projeto. A solução proposta é completamente interoperável, sem o uso de interfaces proprietárias, e totalmente de acordo com as especificações da OMG. Nesse sentido, podemos considerá-la bastante próxima ao que seria uma solução definitiva em termos de comunicação de grupo com ordem total no CORBA. Definimos um algoritmo de difusão atômica para ser implementado, na forma de um *plugin* ETF, dentro ROMIOP como forma de validação da arquitetura proposta, algumas medidas mostrando o custo da ordenação total dentro do ORB também são mostradas.

Este artigo está organizado da seguinte forma: na seção 2 são apresentados alguns trabalhos relacionados. Na seção 3, a arquitetura do MJaco e um conjunto de extensões à especificação ROMIOP é apresentada. Na seção 4, é introduzido o algoritmo de ordenação total utilizado. A seção 5 apresenta algumas considerações a respeito da implementação. Alguns resultados obtidos com o ROMIOP podem ser visualizados na seção 6 e finalmente, a seção 7 apresenta conclusões deste trabalho.

2. Trabalhos Relacionados

Comunicação de grupo no CORBA foi e continua sendo um tema de bastante interesse. Os primeiros trabalhos sobre o assunto utilizavam ferramentas comerciais de comunicação de grupo. Esses trabalhos podem ser identificados na literatura dentro três soluções básicas: as abordagens de integração [Maffeis, 1995], de serviço [Felber, 1998] e de interceptação [Moser et. al., 1998, Lau, 2001].

A abordagem de integração consiste na construção ou na modificação de um ORB existente, adicionando mecanismos de processamento em grupo. A idéia principal nesta abordagem é que o processamento de grupo seja suportado por um sistema de comunicação de grupo abaixo do núcleo do ORB. Já na abordagem usando objetos de serviço é prover o suporte a grupos de objetos como um conjunto de serviços no topo do ORB, e não como parte do próprio ORB. Finalmente, a abordagem de interceptação prevê que as mensagens enviadas aos objetos servidores devem ser capturadas e mapeadas em um sistema de comunicação de grupo, de maneira transparente para a aplicação.

Em [Bessani et. al., 2004] é proposta uma implementação de difusão atômica sobre o MJaco. Esta implementação utiliza os protocolos MIOP e IIOP no desenvolvimento de um sistema ótimo em vários aspectos. Um ponto negativo desse trabalho em relação ao ROMIOP é que ele depende da pesada infra-estrutura do FT-CORBA, implementada através do sistema GroupPac.

A presente proposta tem a virtude de ter disponível as últimas especificações OMG relacionadas à comunicação de grupo. O uso dessas permitiu-nos alcançar a completa interoperabilidade e portabilidade do ORB, requisitos fundamentais de qualquer especificação da OMG.

3. Arquitetura MJaco

O MJaco (http://grouppac.sourceforge.net) é um *middleware* CORBA com suporte para comunicação de grupo baseado nas especificações UMIOP, padronizadas pela OMG.

Este *middleware* permite a difusão de mensagens de forma não confiável, de acordo com o padrão UMIOP, ou confiável, implementados pelos protocolos ReMIOP [Bessani et. al., 2003a] e ROMIOP, todos os três baseados na pilha UDP/*multicast* IP.

Na figura 1, temos o ORB com as duas pilhas de protocolos: uma para a comunicação ponto a ponto, baseada em IIOP, utilizando os serviços do TCP/IP, e outra para a comunicação multiponto formada pelo MIOP e ROMIOP, que utiliza UDP/*multicast* IP. O modelo de integração apresenta os vários elementos definidos na especificação que compõem o suporte para os dois modelos de comunicação.

A primeira fase do projeto do MJaco foi a integração do UMIOP no ORB e a sua implementação. Dando continuidade a este projeto, foi implementado o protocolo ReMIOP, que estende as especificações UMIOP com propriedades de difusão confiável, oferecendo garantias do tipo "melhor esforço" de que mensagens difundidas serão entregues por todos os membros corretos de um grupo. Por fim, o ROMIOP foi adicionado ao rol de protocolos, sendo este último o único que, além de garantir a difusão confiável, atende a garantia de ordenação total das mensagens, ou seja, todos os membros do grupo entregam todas as mensagens na mesma ordem.

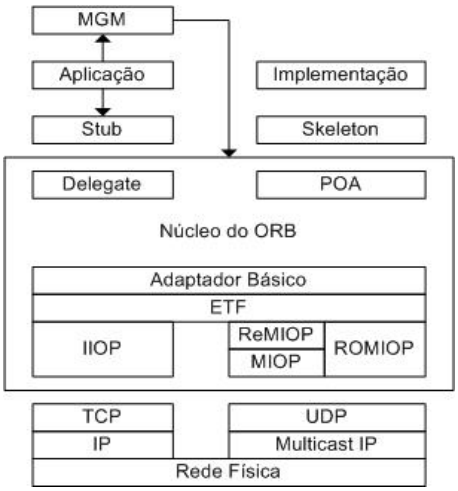


Figura 1: Arquitetura do MJaco.

É importante notar a forma em que o ROMIOP foi introduzido no MJaco. Ao invés de colocá-lo em uma camada acima do ReMIOP, ou até mesmo no mesmo nível do ReMIOP utilizando como base o MIOP, ele foi implementado a partir da base. Isso, como será visto mais adiante, foi feito devido a diferenças consideráveis no formato da especificação desses protocolos, sendo preferível iniciar sua implementação sem tomar como base nada do MIOP.

3.1. A Especificação ETF

A ETF é a especificação de uma plataforma que permite a terceiros projetar e implementar *plugins* de transportes de mensagens. Com ela, os diversos *middlewares* que implementam o CORBA tornam-se muito mais flexíveis devido ao fato de que há na especificação todas as classes e métodos que devem (ou podem, no caso dos opcionais) ser implementados e quais funcionalidades cada um deve ter, tornando desnecessário modificar o código do ORB em si. Isto assegura as propriedades de portabilidade e interoperabilidade do ORB.

O grande problema desta especificação é que ela define apenas como adicionar *plugins* de transporte ponto a ponto, o que a torna deficiente para protocolos multiponto, tal como o ROMIOP. Devido a esse fato, o *middleware* escolhido teve de ser levemente alterado, adicionando a funcionalidade de envio de mensagens para grupos.

Basicamente, a especificação define quatro interfaces obrigatórias: *connection, profile, listener e factories*. A primeira separa a camada de mensagens (GIOP) da camada da ETF, criando um canal de interação entre as mensagens e as conexões (tanto de clientes quanto de servidores). A segunda armazena todas as informações relativas ao protocolo, como métodos para enviar (*marshalling*) e receber informações através da IOR. A terceira provê a iniciativa de "ser conectada" a uma requisição feita por um cliente, direcionando essa requisição para um servidor. A quarta e última interface é a responsável por criar instâncias de clientes, sendo ela a responsável pela ligação do ORB com o *plugin* (a figura 4, na seção 5, apresenta a implementação do ROMIOP como um *plugin* ETF).

3.2. A Especificação ROMIOP

O ROMIOP (atualmente um *draft* [OMG, 2003b]) define um conjunto de interfaces para provisão de comunicação multiponto com garantia de entrega e ordem total para todos os membros não faltosos de um de grupo de objetos. Ele suporta tanto requisições que necessitam de resposta (*two-way requests/replies*) quanto as que não precisam (*oneway requests*). Esta especificação foi projetada para que os protocolos que a implementam coexistam tanto com o IIOP, quanto com MIOP e qualquer outro protocolo de comunicação *multicast*, não podendo interferir no funcionamento destes.

A especificação define uma interface de configuração dos vários possíveis métodos para consolidação de respostas de requisições e qualidade do serviço de ordenação. Em relação ao primeiro fator há dois meios distintos para realizar a consolidação: **votação simples**, onde há três possibilidades para definição da resposta (primeira recebida, última recebida e a primeira que satisfaça o parâmetro de consistência de dados); **votação por quorum**, onde há duas possibilidades para definição da resposta (número de membros que enviam a mesma resposta e porcentagem de membros), ambas dependentes do parâmetro de consistência.

O parâmetro de consistência de dados determina três possibilidades: todas as respostas devem ser iguais; todas as respostas devem ser diferentes (aparentemente sem utilidade); e a padrão, em que a maioria das respostas iguais prevalece. É interessante notar que com essa última opção é possível fornecer um suporte limitado à tolerância de faltas, utilizando-se da idéia de replicação da máquina de estados [Schneider, 1990]. Além disso, há mais um parâmetro relativo à consolidação de respostas que acaba sobrepondo-se a todos os outros. Pode-se configurar um tempo limite para o recebimento das respostas, e mesmo se a opção escolhida ainda não tiver sido atendida, o processo de consolidação é forçado com os resultados que já foram obtidos.

Uma outra interface prevista na especificação é o serviço de grupo, que fornece operações básicas como adição e remoção de membros, além de criação de grupos.

Por fim, em termos de protocolos de comunicação, o ROMIOP define apenas formatos para vários tipos de mensagens. Este formato consiste em um cabeçalho que possui um tamanho fixo e trás informações relativas aos dados contidos no pacote (como seu tipo, número de identificação, posição do pacote dentro de um conjunto de mensagens, dentre outros) e uma área de dados. Os tipos de pacotes definidos são: *request*, *reply*, ACK, NAK, *keepalive*, *cancel* e *memberchange*. Cada tipo possui uma estrutura diferente na área de dados, de acordo com a semântica do pacote. Dentre os

campos podemos citar: para qual membro o pacote é direcionado, endereço de ACK/NAK, ID do pacote já recebido, status do membro, dentre muitos outros.

Cada um dos tipos de pacotes acima citados vem sempre acompanhado inicialmente por um cabeçalho padrão (*PacketHeader*) que informa, além do tipo do pacote seguinte, informações complementares, como se há a necessidade de envio de ACK, o tamanho do identificador único do pacote (que pode variar entre 4 valores distintos: 4, 8, 16 ou 32 bytes), versão do protocolo dentre outras. Abaixo pode ser vista justamente essa definição acima descrita.

```
module ROMIOP {
     typedef octet PacketType;
     const PacketType Request = 0;
     const PacketType Reply = 1;
     const PacketType ACK = 2;
     const PacketType NAK = 3;
     const PacketType KeepAlive = 4;
     const PacketType Cancel = 5;
     const PacketType MemberChange = 6;
     const PacketType MsgOrder = 7;
     struct PacketHeader_1_0 {
        char magic[4];
        octet version;
        octet flags;
        short packet_type;
        unsigned long packet_size;
    }; };
```

3.3. Extensões Propostas no ROMIOP e ETF

Visto que até a presente data, a especificação do ROMIOP ainda não foi concluída, diversas considerações tiveram de ser tomadas e adicionadas para que a implementação desta especificação fosse realizável. A mais importante foi a criação de mais um tipo de mensagem, a que contém a ordem das mensagens enviadas pelo receptor líder (*MsgOrder*). Sua definição é bem simples: há um identificador de quem enviou a mensagem e uma lista com uma estrutura também criada que contém o identificador da mensagem e a identificação do membro emissor que enviou a mensagem.

Outro ponto crucial é em relação à consolidação das respostas. Somente é dito quais modelos devem ser implementados (votação simples ou por quorum), porém não é informado aonde deve ocorrer a consolidação em si (no emissor ou nos receptores). Neste modelo, o algoritmo de consolidação é processado no cliente que requisitou as respostas, tirando dos servidores essa carga extra de processamento. Essa questão pode ser considerada a mais trabalhosa para ser desenvolvida.

Devido ao fato de não haver nenhum módulo relacionado à consolidação de respostas na especificação da ETF, bem como a ausência de material relacionado ao assunto em outros trabalhos, toda uma modelagem teve de ser desenvolvida para solucionar o problema. Diversas abordagens foram analisadas, implementadas e testadas antes de se chegar ao método definitivo que atendesse a especificação incompleta do ROMIOP. Justamente devido a esse fato que o protocolo não utilizou como base nenhum outro previamente existente (ver figura 1 na seção 3). O controle das respostas recebidas não passa pelo protocolo, sendo função do *middleware* definir a quem pertence a resposta. Essa abordagem teve de ser contornada, visto que desse

modo apenas a primeira resposta seria utilizada, descartando todas as outras (fato que ocorreria na existência de mais de 1 servidor). A solução escolhida foi criar um desvio dessa informação para o protocolo do ROMIOP, após o envio da mesma para o *middleware*, atendendo dessa forma ambas as especificações (ETF e ROMIOP).

Apesar de aparentar ser um método lento (a mesma informação passa duas vezes pelo protocolo) os testes realizados (ver seção 6) comprovaram que a perda de performance foi insignificante. Além disso, esse método possui a vantagem de modularizar as funções: mensagens que necessitam ser consolidadas passam por esse processo enquanto as que não, seguem o caminho direto. Dessa forma cada cliente processa a consolidação de suas requisições enquanto os servidores ficam com a tarefa de responder as requisições e definir a ordem das mensagens.

Justamente devido à necessidade de consolidar as respostas é que foi implementado um serviço de membros (*membership*) com mais funcionalidades do que o existente no MJaco. É preciso saber exatamente a quantidade de membros funcionais para que o cliente saiba quantas respostas esperar. Todo esse módulo foi projetado sem nenhum tipo de especificação. Visto que o protocolo necessita manter uma lista com os membros atualizada, e o mesmo também deve ser capaz de lidar com falhas de omissão, falhas de *crash* e problemas com a rede física (cabo de rede cortado, roteador mal configurado), foi implementado um algoritmo executado pelo servidor líder. Nele, a cada intervalo de tempo pré-determinado, é enviada uma mensagem para o grupo perguntando se estão vivos (*alive*). Todos os que não enviarem um ACK para esta mensagem são excluídos do grupo, ou seja, é assumida a abstração de detector perfeito.

Finalmente, a especificação em seu estágio atual não define o algoritmo de ordenação total utilizado. Assim foi definido um algoritmo baseado em ordenador fixo [Défago et. al., 2004], apresentado na seção 4, no sentido de verificar as potencialidades da arquitetura proposta.

4. Algoritmo de Ordenação Total

Um algoritmo de ordenação total com difusão confiável é aquele que garante que todos os processos não-faltosos pertencentes a um grupo entreguem o mesmo conjunto de mensagens na mesma ordem [Défago et. al., 2004]. Esse tipo de algoritmo também é chamado de difusão atômica porque a entrega da mensagem ocorre como se fosse uma primitiva indivisível: a mensagem é entregue para todos ou para nenhum, e se entregue, todas as outras mensagens são ordenadas antes ou depois dessa.

4.1. Modelo de Sistema e Premissas

Em relação a falhas de processos, assumimos um modelo de falhas de parada (*crash*). As falhas de processos são toleradas na medida em que o módulo de gerência de grupos (*membership*) mantém uma lista dos membros atualizada.

Em relação aos canais, assumimos uma semântica de entrega perfeita para mensagens. Esta semântica é implementada a partir da retransmissão periódica das mensagens até que todos os processos receptores acusem o recebimento da mesma (através de uma mensagem ACK). Mensagens duplicadas são detectadas através de seu identificador e descartadas, porém ACKs para estas mensagens são enviados antes de seu descarte uma vez que o ACK pode ter sido perdido pelo seu emissor.

Suposições em relação ao tempo tiveram de ser realizadas para implementar o protocolo do ROMIOP. Para determinar o reenvio das mensagens devido ao não recebimento de ACKs suficientes foi adotado que a soma dos tempos de computação e comunicação é síncrono, ou seja, há um limite superior conhecido para que ambos ocorram. Outra suposição em relação ao tempo tomado foi o de um detector de falha perfeito. Se o tempo levado para chegar a resposta da mensagem perguntando se o processo está vivo (*keepalive*) ultrapassar um certo valor, aquele processo será considerado com problema e será excluído do grupo.

Se algum processo travar e não retornar (*crash*) não há problemas, visto que o módulo de membership mantém uma lista da quantidade de membros atualizada. Uma outra possibilidade é o travamento de um processo e depois seu retorno. Nesse caso, dependendo do tempo que o processo permanecer sem responder ele poderá ser removido do grupo, porém ao retornar, ele será re-incluído na lista de membros. O único problema relacionado a essa última possibilidade é que qualquer resposta oriunda de uma requisição recebida anteriormente ao travamento será descartada pelos receptores.

Por fim, foi implementado um algoritmo de eleição de líder em que o primeiro processo a entrar no grupo é considerado o líder. Isso é realizado enviando-se uma mensagem e esperando por um certo tempo. Se ninguém o responder, ele se considera o líder e avisa a todos que dali pra frente entrarem no grupo da existência do líder. É importante frisar que todos os parâmetros de *timeout* relacionados a tempo são configuráveis para que o protocolo trabalhe da melhor forma possível em cada ambiente de rede.

4.2. Algoritmo

O protocolo adotado para difusão atômica no ROMIOP é baseado no paradigma do ordenador fixo [Défago et al. 2004]. Basicamente, o protocolo funciona da seguinte forma: o emissor difunde uma mensagem solicitando aos membros do grupo sua entrada nesse mesmo grupo. Em seguida, envia sua requisição para o endereço do grupo e fica aguardando uma quantidade de ACKs iguais à quantidade de receptores presentes no grupo. Se a quantidade de ACKs for menor do que o esperado, a requisição é reenviada para o grupo. O algoritmo simplificado do ROMIOP é apresentado na figura 3.

Inicialmente, todos os buffers são inicializados com valores vazios (linha 1). Todas as mensagens a serem enviadas, tanto por clientes quanto servidores, precisam, antes, ser armazenadas (linha 8) e um *timer* precisa ser criado (linha 9) para cada uma delas, menos as do tipo ACK. Isso é necessário, pois todos esses tipos de mensagens necessitam de uma confirmação de que a mensagem enviada efetivamente chegou no destino, sinalizado com o envio de um ACK. Se o ACK não chegar ao emissor da mensagem, essa é reenviada após o término do *timer*. Quando uma quantidade suficiente de ACKs é recebida (linha 42), o vetor que armazenava a mensagem é esvaziado (linha 44), e o *timer* de reenvio da mesma é parado (linha 43).

As requisições (*requests*) recebidas por cada servidor são armazenadas em seu buffer temporário (linha 17). Assim que o receptor líder (o primeiro a entrar no grupo) receber uma requisição qualquer, um *timer* configurável é inicializado. Após o término

desse *timer* é enviado para todos os membros servidores do grupo uma mensagem do tipo *msgOrder* com a ordenação de todas as mensagens de requisição (*request*) recebidas. Todos os servidores ao receberem essa mensagem (linha 33), comparam os identificadores com os armazenados no buffer de mensagens já recebidas (linha 35). Todas as mensagens que o receptor em questão possuir, partindo-se da primeira e indo seqüencialmente até a ultima, serão entregues (delivered, linha 36). Se por alguma razão esse receptor não possuir uma das mensagens, todas as subseqüentes não serão entregues, até que a faltante seja recebida.

```
1.  {Inicialização}
2.  buffer <- ϕ {buffer de mensagens enviadas}
3.  received <- ϕ {buffer de requisições recebidas e não entregues}
4.  order <- ϕ {buffer com a ordem das requisições a serem entregues}
5.  members <- myself.id {lista dos membros do grupo}

6.  {To R-multicast(m)} {difusão de uma mensagem para o grupo}
7.  if m.type = REQUEST | REPLY | MEMBERCHANGE | MSGORDER then {m.type:
    tipo da mensagem m}
8.        buffer <- buffer U {m}
9.        timeout(m, 2000) {inicia um timeout para reenvio se necessário}
10.       send(m) {envia a mensagem}
11. end if
12. else if m.type = ACK then
13.       send(m)
14. end if

15. {To R-receive(m)} {recebimento de uma mensagem}
16. if m.type = REQUEST then
17.       received <- received U {m}
18  end if
19. else if m.type = REPLY then
20.       consolidate(m) {consolida as respostas recebidas}
21. end if
22. else if m.type = MEMBERCHANGE then
23.       if m.status = added then {m.status: tipo da mudança do membro}
24.             members <- members U {m.id}
25.       end if
26.       else if m.status = removed then
27.             members <- members \ {m.id}
28.       end if
29.       else if m.status = online then
30.             R-multicast(members)
31.       end if
32. end if
33. else if m.type = MSGORDER then
34.       order <- order U {m}
35.       while m.id e received.id do
36.             deliver(received) {entrega a mensagem na ordem correta}
37.             order <- order \ m
38.             received <- received \ m
39.       end while
40. end if
41. else if m.type = ACK then
42.       if enoughACK(m) = OK then {se recebeu quantidade suficiente de
    ACKs}
43.             timeout(m).stop {pára o processo que reenvia mensagem}
44.             buffer <- buffer \ m
45.       end if
46. end if
```

Figura 3: Algoritmo simplificado do ROMIOP.

Logo após o processamento da mensagem pelo servidor, se a mensagem necessitar de retorno (*two-way*), uma mensagem de resposta (*reply*) será enviada para o endereço do grupo (linha 7). O cliente emissor que enviou a mensagem de request permanece aguardando uma certa quantidade de respostas pré-determinadas, dependendo de como a consolidação foi configurada (linha 19).

Por fim, sempre que um objeto deseja entrar no grupo, inicialmente é enviada uma mensagem com o status *added*. Todos os membros que recebem essa mensagem com esse status adicionam o membro a sua lista de membros (linha 23). Após o membro receber a confirmação de que entrou no grupo, ele envia uma mensagem solicitando a lista atualizada de membros, indicando-a com o status *online*. Todos os membros, ao receber essa mensagem (linha 29), retornam a sua lista de membros do grupo.

5. Implementação do ROMIOP sobre a ETF

Para melhor apresentar o protocolo e a forma de implementação que foi adotada para satisfazer os requisitos da especificação ETF, segue na figura 4, um diagrama de classes apenas com os métodos e atributos extremamente essenciais, além de serem mostradas apenas as relações entre classes de importância vital para o correto entendimento.

Basicamente, na chegada de qualquer pacote de mensagem, através da classe ROMIOPConnection , o fragmento, após diversas verificações, é adicionado a um objeto da classe FragmentedMessage . Quando uma mensagem estiver completa ela então é entregue ao algoritmo do ROMIOP, para realização da ordenação.

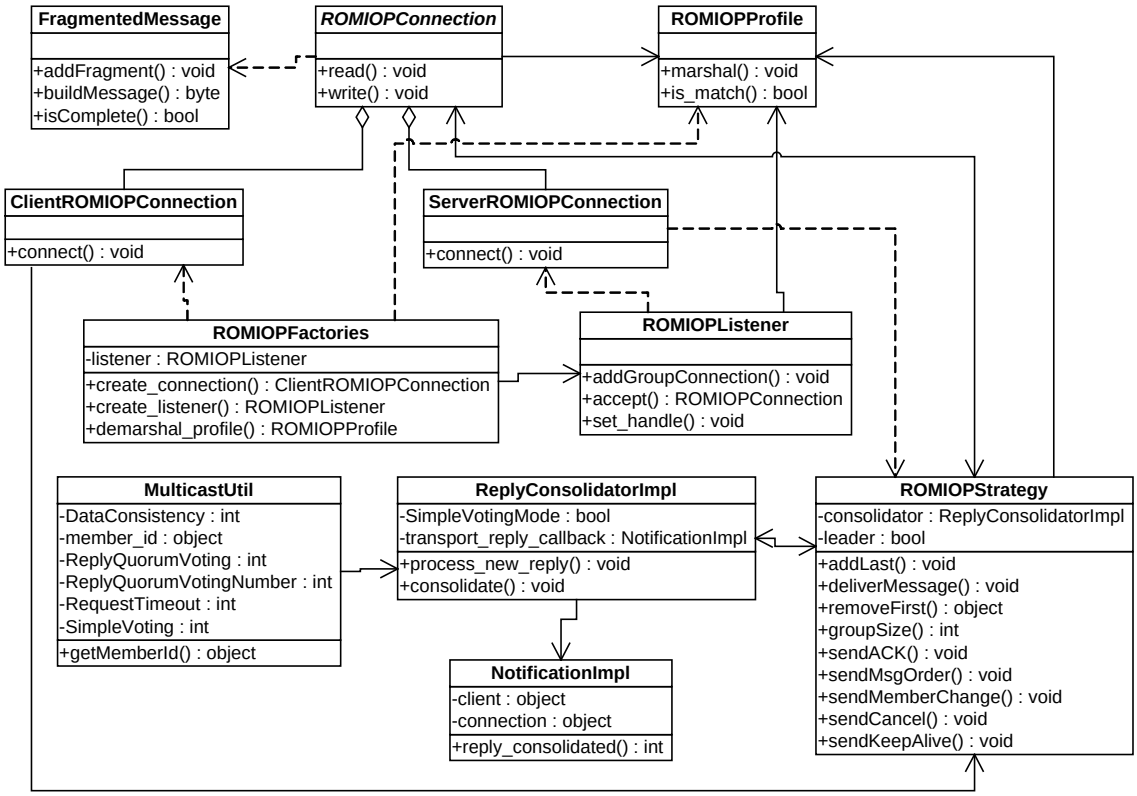


Figura 4: Diagrama de classes simplificado do *plugin* ROMIOP.

Praticamente todas as classes do protocolo utilizam-se das classes base MultciastUtil e ROMIOPProfile , sendo cada uma responsável por guardar

informações específicas. A primeira relaciona-se com todo o protocolo, guardando configurações do protocolo, como método de consolidação e tempos limites. A segunda está unicamente ligada a uma conexão, sendo responsável por guardar informações desta, como o endereço do grupo.

Para mensagens que necessitam de resposta, as classes ReplyConsolidatorImpl e NotificationImpl são utilizadas, sendo a segunda controlada pela primeira e esta controlada pela classe ROMIOPStrategy . Enquanto a classe ReplyConsolidatorImpl é acionada na criação da mensagem que necessita de resposta, bem como na chegada de qualquer uma das respostas, além de efetivamente consolidar a resposta, a classe NotificationImpl só é usada para notificar o ORB da resposta escolhida. Por fim, a classe ROMIOPStrategy mantém todos os outros processos necessários para o correto funcionamento do protocolo, sendo por isso considerada a mais complexa e com mais funcionalidades. Algumas de suas funções estão o envio dos ACKs, controle de membership e envio da ordem das mensagens.

6. Avaliação de Desempenho

Com o intuito de analisar o desempenho MJaco, bem como as opções feitas na implementação do mesmo, foram realizados diversos testes. O ambiente em que os testes foram realizados foi um conjunto de máquinas rodando o Windows XP. O cliente foi executado em uma máquina Athlon 2600+, com 512MB de memória RAM. O servidor líder foi executado em um Pentium 4 de 2.6GHz com 1,5GB de memória RAM. Os outros dois servidores foram executados em máquinas Athlon de 1.47GHz com 248MB de memória RAM cada. O teste a seguir foi realizado para analisar o algoritmo de consolidação, onde foi comparada a opção atômica (onde o cliente aguarda a resposta de cada servidor) com a de primeira resposta. A figura 5 abaixo mostra os resultados com 2 e com 3 servidores.

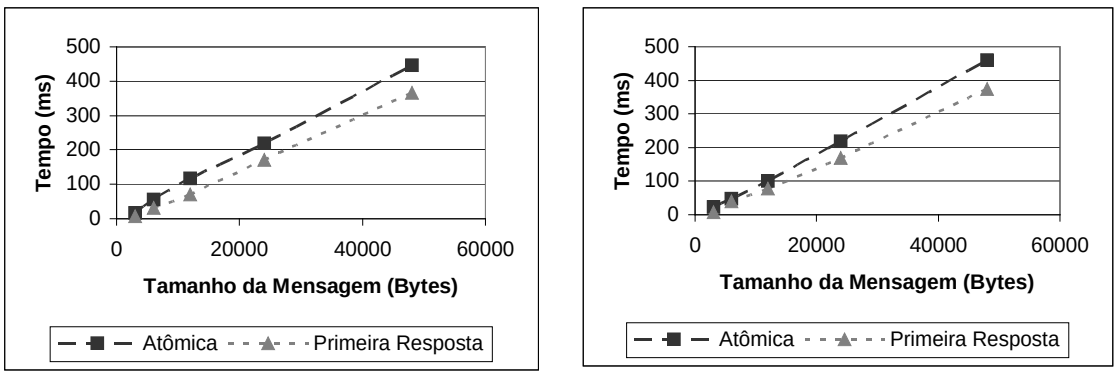


Figura 5: Comparação de dois tipos de consolidação com 2 e 6 servidores.

A análise dos gráficos trás, além do esperado que o tempo de consolidar somente a primeira resposta ser menor que o de todas, o fato de que para pequenas mensagens (aproximadamente até 3000 bytes) não há praticamente qualquer ganho de performance, sendo nesse caso mais vantajoso usar a consolidação atômica, que trás mais segurança. Com esse resultado, nota-se que para grandes tamanhos de mensagens é crucial a escolha do método de consolidação: se a aplicação necessita suportar faltas bizantinas ou apenas deseja-se ter certeza da resposta obtida (como sistemas de suporte a vida ou

aplicações militares) haverá um custo considerável. É importante notar que mesmo com o incremento do número de servidores, o tempo para se obter a resposta oriunda da consolidação via primeira resposta praticamente não aumenta.

7. Conclusão

Este artigo apresentou um estudo sobre a implementação da especificação preliminar ROMIOP utilizando os princípios da especificação ETF. A primeira especificação visa padronizar interfaces e formatos de mensagens para difusão de mensagens com garantias como confiabilidade e ordenação total, enquanto a segunda define métodos para integração de novos protocolos em sistemas já existentes. A partir da finalização desta especificação, teremos finalmente mecanismos interoperáveis de comunicação de grupo em *middleware* aberto (padronizado). Com o ROMIOP agora concluído é possível analisar cada módulo minuciosamente em busca de um melhor desempenho tornando a implementação mais eficiente e com mais funcionalidades.

Mais informações relativas à testes de desempenho, ao próprio algoritmo desenvolvido, bem como códigos fonte podem ser encontradas na internet, na página web do grupo de desenvolvimento (http://grouppac.sourceforge.net).

Referências Bibliográficas

Bessani A. N., Fraga J. S., Lung L. C. (2003a). ReMIOP: Projeto e Implementação de um Mecanismo de Difusão Confiável no CORBA. Anais do XXI SBRC 03, Natal RN.

Bessani A. N., Fraga J. S., Lung L. C., et. al.(2003b). Integrating the unreliable multicast inter-orb protocol in mjaco. In Proc. of the 4th IFIP WG 6.1 Inter. Conf. on Distributed Applications and Interoperable Systems - IFIP DAIS 03, LNCS v2893, Paris - France.

Bessani, A. N., Fraga, J. S., Lung, L. C., Alchieri, E. A. B (2004). Active Replication in CORBA: Standards, Protocols and Implementation Framework. In: 6th Int. Symposium on Distributed Objects and Applications, 2004, Larnaca, Cyprus. Proc. of DOA'04.

Deering, S. E. (1986). Host extensions for ip multicasting (rfc 988). IETF RFC.

Défago X., Schiper A. e Urbán P. (2004) Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372-421, Dec. 2004. ACM Press.

Felber, P. (1998), "The CORBA Object Group Service A Service Approach to Object Groups in CORBA", PhD. Thesis, École Polytechnique Fédérale de Lausanne, Lausanne,

Lau C. L., J. S. Fraga, J. Farines, M. Ogg, A. Ricciardi, CosNamingFT A Fault-Tolerant CORBA Naming Service 18th IEEE Symposium on Reliable Distributed Systems SRDS 99, Lausanne, Suiça, October 1999.

Maffeis, S., Run-Time Support for Object-Oriented Distributed Programming, Ph.D. Thesis University of Zurich. Zurich, 1995.

Moser, L. E., P. M. P. Melliar-Smith, Narasimhan, P., Consistent Object Replication in the Eternal System, Theory and Practice of Object Systems, 4(2): 81-92, 1998.

OMG (2002). The Common Object Request Broker Architecture v3.0. OMG Doc. 02-06-33.

OMG (2000).Object Management Group, Fault-Tolerant CORBA Specification V1.0, OMG document: ptc/2000-04-04, April, 2000. www.omg.org.

OMG (2002a). *The Common Object Request Broker Architecture* v3.0. OMG Doc. 02-06-33.

OMG (2003). Extensible transport framework specification v1.0. OMG Standard.

OMG (2003a). Unreliable multicast inter-orb protocol spec v1.0. OMG Doc. ptc/03-01-11.

OMG (2003b). Reliable, ordered, multicast inter-orb protocol. Revised Submission OMG Document realtime/2003-10-04. October, 2003.

Schneider, F. B. (1990) Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4): 299-314, Dec. 1990. ACM Press.