XXVIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos
24 a 28 de maio de 2010
Gramado, RS

XI Workshop de Testes e Tolerância a Falhas (WTF)

Editora
Sociedade Brasileira de Computação (SBC)

Organizadores
Fernando Luís Dotti (PUCRS)
Antônio Jorge Gomes Abelém (UFPA)
Luciano Paschoal Gaspary (UFRGS)
Marinho Pilla Barcellos (UFRGS)

Realização
Instituto de Informática
Universidade Federal do Rio Grande do Sul (UFRGS)

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

Capa: Josué Klafke Sperb
Produção Editorial: Flávio Roberto Santos, Roben Castagna Lunardi, Matheus Lehmann, Rafael Santos Bezerra, Luciano Paschoal Gaspary e Marinho Pilla Barcellos.

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

Dados Internacionais de Catalogação na Publicação (CIP)

Workshop de Testes e Tolerância a Falhas (11. : 2010 : Gramado, RS).

Anais / XI Workshop de Testes e Tolerância a Falhas; organizadores Fernando Luís Dotti... et al. − Porto Alegre : SBC, c2010.

177 p.

ISSN 2177-496X

1. Redes de computadores. 2. Sistemas distribuídos. I. Dotti, Fernando Luís. II. Título.

Promoção

Sociedade Brasileira de Computação (SBC)

Diretoria

Presidente
José Carlos Maldonado (USP)

Vice-Presidente
Marcelo Walter (UFRGS)

Diretor Administrativo
Luciano Paschoal Gaspary (UFRGS)

Diretor de Finanças
Paulo Cesar Masiero (USP)

Diretor de Eventos e Comissões Especiais
Lisandro Zambenedetti Granville (UFRGS)

Diretora de Educação
Mirella Moura Moro (UFMG)

Diretora de Publicações
Karin Breitman (PUC-Rio)

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

Diretora de Secretarias Regionais
Thais Vasconcelos Batista (UFRN)

Diretor de Divulgação e Marketing
Altigran Soares da Silva (UFAM)

Diretor de Regulamentação da Profissão
Ricardo de Oliveira Anido (UNICAMP)

Diretor de Eventos Especiais
Carlos Eduardo Ferreira (USP)

Diretor de Cooperação com Sociedades Científicas
Marcelo Walter (UFRGS)

Promoção

Conselho

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

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

Suplentes - Mandato 2009-2011
Geraldo B. Xexeo (UFRJ)
Taisy Silva Weber (UFRGS)
Marta Lima de Queiroz Mattoso (UFRJ)
Raul Sidnei Wazlawick (UFSC)
Renata Vieira (PUCRS)

Laboratório Nacional de Redes de Computadores (LARC)

Diretoria

Diretor do Conselho Técnico-Científico
Artur Ziviani (LNCC)

Diretor Executivo
Célio Vinicius Neves de Albuquerque (UFF)

Vice-Diretora do Conselho Técnico-Científico
Flávia Coimbra Delicato (UFRN)

Vice-Diretor Executivo
Luciano Paschoal Gaspary (UFRGS)

Membros Institucionais
CEFET-CE, CEFET-PR, IME, INPE/MCT, LNCC, PUCPR, PUC-RIO, SESU/MEC, UECE, UERJ, UFAM, UFBA, UFC, UFCG, UFES, UFF, UFMG, UFPA, UFPB, UFPE, UFPR, UFRGS, UFRJ, UFRN, UFSC, UFSCAR, UNICAMP, UNIFACS, USP.

Realização

Comitê de Organização

Coordenação Geral
Luciano Paschoal Gaspary (UFRGS)
Marinho Pilla Barcellos (UFRGS)

Coordenação do Comitê de Programa
Luci Pirmez (UFRJ)
Thaís Vasconcelos Batista (UFRN)

Coordenação de Palestras e Tutoriais
Lisandro Zambenedetti Granville (UFRGS)

Coordenação de Painéis e Debates
José Marcos Silva Nogueira (UFMG)

Coordenação de Minicursos
Carlos Alberto Kamienski (UFABC)

Coordenação de Workshops
Antônio Jorge Gomes Abelém (UFPA)

Coordenação do Salão de Ferramentas
Nazareno Andrade (UFCG)

Comitê Consultivo
Artur Ziviani (LNCC)
Carlos André Guimarães Ferraz (UFPE)
Célio Vinicius Neves de Albuquerque (UFF)
Francisco Vilar Brasileiro (UFCG)
Lisandro Zambenedetti Granville (UFRGS)
Luís Henrique Maciel Kosmalski Costa (UFRJ)
Marcelo Gonçalves Rubinstein (UERJ)
Nelson Luis Saldanha da Fonseca (UNICAMP)
Paulo André da Silva Gonçalves (UFPE)

Realização

Organização Local
Adler Hoff Schmidt (UFRGS)
Alan Mezzomo (UFRGS)
Alessandro Huber dos Santos (UFRGS)
Bruno Lopes Dalmazo (UFRGS)
Carlos Alberto da Silveira Junior (UFRGS)
Carlos Raniery Paula dos Santos (UFRGS)
Cristiano Bonato Both (UFRGS)
Flávio Roberto Santos (UFRGS)
Jair Santanna (UFRGS)
Jéferson Campos Nobre (UFRGS)
Juliano Wickboldt (UFRGS)
Leonardo Richter Bays (UFRGS)
Lourdes Tassinari (UFRGS)
Luís Armando Bianchin (UFRGS)
Luis Otávio Luz Soares (UFRGS)
Marcos Ennes Barreto (UFRGS)
Matheus Brenner Lehmann (UFRGS)
Pedro Arthur Pinheiro Rosa Duarte (UFRGS)
Pietro Biasuz (UFRGS)
Rafael Pereira Esteves (UFRGS)
Rafael Kunst (UFRGS)
Rafael Santos Bezerra (UFRGS)
Ricardo Luis dos Santos (UFRGS)
Roben Castagna Lunardi (UFRGS)
Rodolfo Stoffel Antunes (UFRGS)
Rodrigo Mansilha (UFRGS)
Weverton Luis da Costa Cordeiro (UFRGS)

Mensagem dos Coordenadores Gerais

Bem-vindo(a) ao XXVIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC 2010)! Esta edição do simpósio está sendo realizada de 24 a 28 de maio de 2010 na pitoresca cidade de Gramado, RS. Promovido pela Sociedade Brasileira de Computação (SBC) e pelo Laboratório Nacional de Redes de Computadores (LARC) desde 1983, o SBRC 2010 almeja não menos que honrar com uma tradição de quase 30 anos: ser reconhecido como o mais importante evento científico em redes de computadores e sistemas distribuídos do país, e um dos mais concorridos em Informática. Mais do que isso, pretende estimular intercâmbio de idéias e discussões qualificadas, aproximá-lo(a) de temas de pesquisa efervescentes e fomentar saudável aproximação entre estudantes, pesquisadores, professores e profissionais.

Para atingir os objetivos supracitados, reunimos um grupo muito especial de professores atuantes em nossa comunidade que, com o nosso apoio, executou com êxito a tarefa de construir um **Programa Técnico** de altíssima qualidade. O SBRC 2010 abrange as seguintes atividades: 20 sessões técnicas de artigos completos, cobrindo uma grande gama de problemas em redes de computadores e sistemas distribuídos; 2 sessões técnicas para apresentações de ferramentas; 5 minicursos ministrados de forma didática, por professores da área, sobre temas atuais; 3 palestras e 3 tutoriais sobre tópicos de pesquisa avançados, apresentados por especialistas nacionais e estrangeiros; e 3 painéis versando sobre assuntos de relevância no momento. Completa a programação técnica a realização de 8 *workshops* satélites em temas específicos: WRNP, WGRS, WTR, WSE, WTF, WCGA, WP2P e WPEIF. Não podemos deixar de ressaltar o **Programa Social**, organizado em torno da temática "*vinho*", simbolizando uma comunidade de pesquisa madura e que, com o passar dos anos, se aprimora e refina cada vez mais.

Além da ênfase na qualidade do programa técnico e social, o SBRC 2010 ambiciona deixar, como marca registrada, seu esforço na busca por excelência organizacional. Tal tem sido perseguido há mais de dois anos e exigido muita determinação, dedicação e esforço de uma equipe afinada de organização local, composta por estudantes, técnicos administrativos e professores. O efeito desse esforço pode ser percebido em elementos simples, mas diferenciais, tais como uniformização de datas de submissão de trabalhos, portal *sempre* atualizado com as últimas informações, comunicação sistemática com potenciais participantes e pronto atendimento a qualquer dúvida. O nosso principal objetivo com essa iniciativa foi e continua sendo oferecer uma elevada *qualidade de experiência* a você, colega participante!

Gostaríamos de agradecer aos membros do Comitê de Organização Geral e Local que, por conta de seu trabalho voluntário e incansável, ajudaram a construir um evento que julgamos de ótimo nível. Gostaríamos de agradecer, também, à SBC, pelo apoio prestado ao longo das muitas etapas da organização, e aos patrocinadores, pelo incentivo à divulgação de atividades de pesquisa conduzidas no País e pela confiança depositada neste fórum. Por fim, nossos agradecimentos ao Instituto de Informática da UFRGS, por viabilizar a realização, pela quarta vez, de um evento do porte do SBRC.

Sejam bem-vindos à Serra Gaúcha para o "SBRC do Vinho"! Desejamos que desfrutem de uma semana agradável e proveitosa!

Luciano Paschoal Gaspary
Marinho Pilla Barcellos
Coordenadores Gerais do SBRC 2010

Mensagem do Coordenador do WTF

O Workshop de Testes e Tolerância a Falhas (WTF) é um evento anual promovido pela Comissão Especial de Sistemas Tolerantes a Falhas (CE-TF) da Sociedade Brasileira de Computação (SBC). Esta edição acontece em Gramado, RS, em conjunto com o XXVIII Simpósio Brasileiro de Redes de Computadores (SBRC). Me permito, em nome da CE-TF e do Comitê de Programa, saudar, com muita satisfação, a todos participantes da XI edição do WTF.

O WTF cumpre o objetivo de integrar os pesquisadores das áreas de Tolerância a Falhas e Testes, promovendo a discussão de contribuições científicas devotadas à construção de sistemas confiáveis. O Comitê de Programa do WTF 2010 contou com 39 membros atuantes na área, sendo 31 no Brasil, 5 em Portugal, 1 na França, 1 nos Estados Unidos, 1 na Inglaterra e 1 na Suíça. É com satisfação que agradeço a todos membros deste comitê pelo criterioso trabalho de revisão dos artigos.

Em 2010, o WTF contou com 24 submissões, sendo 22 destas artigos completos e 2 resumos estendidos. Excetuando um artigo francês, todos demais trabalhos são oriundos de instituições brasileiras. Tivemos 61 autores, de 22 instituições diferentes, envolvidos nos artigos submetidos. Agradeço enfaticamente a todos autores pela consideração do WTF para submissão.

Destes trabalhos, 12 artigos completos foram selecionados para publicação e apresentação. Os artigos aceitos envolvem 31 autores de 10 instituições diferentes, estando organizados em 4 sessões técnicas: Algoritmos Distribuídos (4 artigos), Injeção de Falhas (2 artigos), Testes e Sistemas Embarcados (4 artigos), e Serviços Web e Roteamento (2 artigos). Para promover o debate e racionalizar o uso do tempo, as seções contam com um espaço único de perguntas aos apresentadores. Além destas seções técnicas, faz parte da programação do WTF também uma Palestra Convidada, a ser ministrada pelo Prof. Henrique Madeira, da Universidade de Coimbra, ao qual agradeço pelo pronto interesse em contribuir com o WTF.

Desejando um ótimo workshop a todos, repito os agradecimentos aos autores de artigos submetidos, ao palestrante convidado e aos membros do comitê de programa, e expresso meus agradecimentos à CE-TF pelas sugestões e constante apoio, e à organização do SBRC pelo excelente trabalho e por todo suporte prestado.

Aproveitem o WTF 2010 e a pitoresca cidade de Gramado!

Fernando Luís Dotti
Coordenador do WTF 2010

Comitê de Programa do WTF

Alcides Calsavara, Pontifícia Universidade Católica do Paraná (PUCPR)
Alexandre Sztajnberg, Universidade do Estado do Rio de Janeiro (UERJ)
Alysson Bessani, Universidade de Lisboa
Ana Ambrosio, Instituto Nacional de Pesquisas Espaciais (INPE)
Avelino Zorzo, Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Cecilia Rubira, Universidade Estadual de Campinas (UNICAMP)
Daniel Mossé, University of Pittsburgh
Eduardo Bezerra, Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Eliane Martins, Universidade Estadual de Campinas (UNICAMP)
Elias P. Duarte Jr., Universidade Federal do Paraná (UFPR)
Fabíola Greve, Universidade Federal da Bahia (UFBA)
Fernando Luís Dotti, Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Fernando Pedone, University of Lugano
Flávio Assis Silva, Universidade Federal da Bahia (UFBA)
Francisco Brasileiro, Universidade Federal de Campina Grande (UFCG)
Frank Siqueira, Universidade Federal de Santa Catarina (UFSC)
Henrique Madeira, University of Coimbra
Irineu Sotoma, Universidade Federal de Mato Grosso do Sul (UFMS)
Ingrid Jansch-Pôrto, Universidade Federal do Rio Grande do Sul (UFRGS)
Joni da Silva Fraga, Universidade Federal de Santa Catarina (UFSC)
Jose Pereira, Universidade do Minho
Lau Cheuk Lung, Universidade Federal de Santa Catarina (UFSC)
Luciana Arantes, Université de Paris VI
Luiz Nacamura Júnior, Universidade Tecnológica Federal do Paraná (UTFPR)
Luiz Carlos Albini, Universidade Federal do Paraná (UFPR)
Luiz Eduardo Buzato, Universidade Estadual de Campinas (UNICAMP)
Marcia Pasin, Universidade Federal de Santa Maria (UFSM)
Marcos Aguilera, Microsoft Research
Miguel Correia, Universidade de Lisboa
Patricia Machado, Universidade Federal de Campina Grande (UFCG)
Raimundo Barreto, Universidade Federal do Amazonas (UFAM)
Raimundo José de Araújo Macêdo, Universidade Federal da Bahia (UFBA)
Raul Ceretta Nunes, Universidade Federal de Santa Maria (UFSM)
Regina Moraes, Universidade Estadual de Campinas (UNICAMP)
Rogerio de Lemos, University of Kent
Rui Oliveira, Universidade do Minho
Sérgio Gorender, Universidade Federal da Bahia (UFBA)
Taisy Weber, Universidade Federal do Rio Grande do Sul (UFRGS)
Udo Fritzke Jr., Pontifícia Universidade Católica de Minas Gerais (PUC Minas)
Vidal Martins, Pontifícia Universidade Católica do Paraná (PUCPR)

Revisores do WTF

Alcides Calsavara, Pontifícia Universidade Católica do Paraná (PUCPR)
Alexandre Sztajnberg, Universidade do Estado do Rio de Janeiro (UERJ)
Alysson Bessani, Universidade de Lisboa
Ana Ambrosio, Instituto Nacional de Pesquisas Espaciais (INPE)
André W. Valenti, Universidade Estadual de Campinas (UNICAMP)
Avelino Zorzo, Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Cecilia Rubira, Universidade Estadual de Campinas (UNICAMP)
Daniel Mossé, University of Pittsburgh
Eduardo Bezerra, Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Eliane Martins, Universidade Estadual de Campinas (UNICAMP)
Elias P. Duarte Jr., Universidade Federal do Paraná (UFPR)
Everton Alves, Universidade Estadual de Campinas (UNICAMP)
Fabíola Greve, Universidade Federal da Bahia (UFBA)
Fernando Luís Dotti, Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Fernando Pedone, University of Lugano
Flávio Assis Silva, Universidade Federal da Bahia (UFBA)
Francisco Brasileiro, Universidade Federal de Campina Grande (UFCG)
Frank Siqueira, Universidade Federal de Santa Catarina (UFSC)
Henrique Madeira, University of Coimbra
Irineu Sotoma, Universidade Federal de Mato Grosso do Sul (UFMS)
Ingrid Jansch-Pôrto, Universidade Federal do Rio Grande do Sul (UFRGS)
Jim Lau, Universidade Federal de Santa Catarina (UFSC)
Joni da Silva Fraga, Universidade Federal de Santa Catarina (UFSC)
Jose Pereira, Universidade do Minho
Lau Cheuk Lung, Universidade Federal de Santa Catarina (UFSC)
Luciana Arantes, Université de Paris VI
Luiz Nacamura Júnior, Universidade Tecnológica Federal do Paraná (UTFPR)
Luiz Carlos Albini, Universidade Federal do Paraná (UFPR)
Luiz Eduardo Buzato, Universidade Estadual de Campinas (UNICAMP)
Marcia Pasin, Universidade Federal de Santa Maria (UFSM)
Marcos Aguilera, Microsoft Research
Miguel Correia, Universidade de Lisboa
Patricia Machado, Universidade Federal de Campina Grande (UFCG)
Patricia Pitthan de Araújo Barcelos, Universidade Federal de Santa Maria (UFSM)
Raimundo Barreto, Universidade Federal do Amazonas (UFAM)
Raimundo José de Araújo Macêdo, Universidade Federal da Bahia (UFBA)
Raul Ceretta Nunes, Universidade Federal de Santa Maria (UFSM)
Regina Moraes, Universidade Estadual de Campinas (UNICAMP)
Rogerio de Lemos, University of Kent
Rui Oliveira, Universidade do Minho
Sérgio Gorender, Universidade Federal da Bahia (UFBA)
Taisy Weber, Universidade Federal do Rio Grande do Sul (UFRGS)
Udo Fritzke Jr., Pontifícia Universidade Católica de Minas Gerais (PUC Minas)
Vidal Martins, Pontifícia Universidade Católica do Paraná (PUCPR)

Sumário

XI Workshop de Testes e Tolerância a Falhas

♦

Sessão Técnica 1

Algoritmos Distribuídos

Consenso com Recuperação no Modelo *Partitioned Synchronous*

Sérgio Gorender e Raimundo Macêdo

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

{macedo,gorender}@ufba.br

Abstract. *The partioned synchronous distributed system model has been introduced to take advantage of synchronous partitions of hybrid distributed systems, as such synchronous partitions are implementable in many real scenarios. In this paper we present for the first time a consensus algorithm for proceses that can recover, and its formal proofs, devoted to the partioned synchronous model. The main advantage of the proposed algorithm is that it can tolerate up to n-k process failures in a system with n processes and k synchronous partitions - not all processes need to belong to synchronous partitions. In particular, such robustness is valid even if the majority of processes does not belong to synchronous partitions, which is an advantage in terms of robustness when compared with algorithms for conventional distributed system models.*

Resumo. *O modelo síncrono particionado (partioned synchronous foi introduzido para tirar proveito de partições síncronas em sistemas distribuídos híbridos, uma vez que estas são implementáveis em muitos cenários reais. No presente artigo apresentamos pela primeira vez um algoritmo para consenso com recuperação de processos, e respectivas provas formais, adequado ao modelo síncrono particionado. O algoritmo proposto tem como principal vantagem a capacidade de tolerar n-k defeitos de processos, onde k é o número de partições síncronas e n o tonal de processos no sistema - sendo que podem existir processos que não fazem parte de partições síncronas. Em particular, a robustez do protocolo se aplica mesmo se a maioria dos processos não estiver em partições síncronas, o que representa uma vantagem em termos de robustez quando comparado com soluções para modelos convencionais.*

1. Introdução

Os sistemas distribuídos são compostos por um conjunto de processos residentes em diversos computadores de uma rede de comunicação, onde os processos se comunicam por troca de mensagens. Uma das principais vantagens dos sistemas distribuídos é a possibilidade de se implementar aplicações tolerantes a falhas, por exemplo, através da replicação de processos, garantindo a continuidade do serviço sendo executado mesmo que ocorram defeitos em um determinado número de processos e canais de comunicação. A capacidade de resolver problemas de tolerância a falhas em sistemas distribuídos está intimamente ligada à existência de um modelo de sistema adequado onde se possa demonstrar a possibilidade de solução de

tais problemas. Portanto, há algumas décadas, pesquisadores vêm propondo uma variedade de modelos de resolução de problemas, onde os modelos assíncronos (ou livres de tempo) e síncronos (baseados no tempo) têm dominado o centro das atenções, por serem considerados modelos extremos em termos de resolução de problemas de tolerância a falhas. Por exemplo, o problema de difusão confiável - na presença de canais confiáveis e falhas silenciosas de processos - é solúvel em ambos os modelos [Lynch 1996]. Contudo, o problema de consenso distribuído é solúvel no modelo síncrono, mas não no modelo assíncrono [Fisher et al. 1985].

A impossibilidade relativa aos sistemas assíncronos levou à pesquisa de modelos alternativos, onde o consenso distribuído pode também ser garantido. Um desses modelos mais utilizados é o parcialmente síncrono, que assume que o "comportamento síncrono" se estabelece durante períodos de tempo suficientemente longos para a execução do consenso [Dwork et al. 1988] (tal propriedade é chamada de *Global Stabilization Time*). Num outro modelo, chamado de detectores de defeitos não confiáveis, a propriedade *Global Stabilization Time* é necessária para garantir que os protocolos de consenso baseados no detector de defeitos $\diamond\mathcal{S}$ funcionem de forma adequada [Chandra and Toueg 1996]. Já o modelo assíncrono temporizado depende de períodos de estabilidade síncrona suficientemente longos para prover serviços [Cristian and Fetzer 1999].

Todos esses modelos de sistema acima são caracterizados por configurações homogêneas e estáticas no que toca os aspectos temporais. Ou seja, uma vez definidas as características temporais dos processos e canais de comunicação, essas não se modificam durante a vida do sistema (estática) e todos os processos e canais de comunicação são definidos com as mesmas características temporais (homogêneo). Uma das exceções no aspecto homogêneo é o sistema TCB [Veríssimo and Casimiro 2002], onde um o sistema assíncrono é equipado com componentes síncronas que formam uma *spanning tree* de comunicação síncrona, ou *wormholes*. No entanto, os modelos baseados em *wormholes* são estáticos em relação às mudanças de qualidade de serviços das componentes síncronas.

Para lidar com aspectos dinâmicos e híbridos de modelos de sistemas distribuídos, atendendo às demandas dos novos ambientes com qualidades de serviço variadas, modelos híbridos e dinâmicos foram introduzidos em [Gorender and Macêdo 2002, Macêdo et al. 2005, Macêdo 2007, Gorender et al. 2007, Macêdo and Gorender 2009]. Em [Gorender and Macêdo 2002], foi apresentado um algoritmo de consenso que requer uma *spanning tree* síncrona no sistema distribuído, onde processos são síncronos e canais de comunicação podem ser síncronos ou assíncronos. Nos trabalhos [Macêdo et al. 2005, Gorender et al. 2007], o requisito de *spanning tree* síncrona foi removido e apresentado soluções para o consenso uniforme em ambientes dinâmicos. Em [Macêdo 2007], o modelo foi generalizado para que processos e canais de comunicação pudessem variar entre síncrono e assíncrono e foi apresentado um algoritmo de comunicação em grupo capaz de ligar com ambientes híbridos e dinâmicos em [Macêdo 2007, Macêdo and Freitas 2009], e finalmente, em [Macêdo and Gorender 2008, Macêdo and Gorender 2009] foi introduzido o modelo híbrido e dinâmico *partitioned synchronous* que requer menos garantias temporais do que o modelo síncrono e onde foi provado ser possível a implementação de detectores perfeitos (mecanismo fundamental para a solução de consenso). Vale salientar que a implementação de detectores perfeitos no modelo *partitioned synchronous* não requer a existência de um *wormhole* síncrono [Veríssimo and Casimiro 2002] ou *spanning tree* síncrona [Gorender and Macêdo 2002], onde seria possível implementar ações síncronas globais em

todos os processos, como sincronização interna de relógios. No sistema *partitioned synchronous*, proposto, componentes do ambiente distribuído necessitam ser síncronos, mas os mesmos não precisam estar conectados entre si via canais síncronos, o que torna impossível a execução de ações síncronas distribuídas em todos os processos do sistema. E mesmo que parte dos processos não esteja em qualquer das componentes síncronas, pode-se ainda assim tirar proveito das partições síncronas existentes para melhorar a robustez das aplicações de tolerância a falhas.

Neste artigo, exploramos o modelo *partitioned synchronous* para propor uma solução robusta para o consenso distribuído. Para isso apresentamos e provamos a correção de um algoritmo para consenso com recuperação de defeitos de processos. Nosso algoritmo se baseia no algoritmo Paxos projetado por Leslie Lamport [Lamport 1998] aplicado ao modelo *partitioned synchronous*. Para sua terminação, o algoritmo proposto não depende da propriedade *Global Stabilization Time* característica dos sistemas parcialmente síncronos; portanto, não dependendo de estabilidade do ambiente em questão, desde que o modelo subjacente seja *partitioned synchronous*. O algoritmo proposto tem como principal vantagem a capacidade de tolerar n-k defeitos de processos, onde k é o número de partições síncronas e n o total de processos no sistema - sendo que podem existir processos que não fazem parte de partições síncronas. Em particular, a robustez do protocolo se aplica mesmo se a maioria dos processos não estiver em partições síncronas, o que representa uma vantagem em termos de robustez quando comparado com soluções para modelos convencionais.

O estudo de soluções de tolerância a falhas para o modelo *partitioned synchronous* tem interesse prático uma vez que muitas configurações reais incluem componentes síncronas, como, por exemplo, processos em *clusters* locais que se comunicam com processos clientes através de redes de longa distância (WAN).

O restante deste artigo está estruturado da seguinte forma. Na seção 2 discutimos trabalhos correlatos. Na seção 3 fazemos uma breve apresentação do modelo *partitioned synchronos* introduzido em [Macêdo and Gorender 2009]. Na seção 4 é apresentado o algoritmo de consenso com recuperação e as respectivas provas de correção. Finalmente, na seção 5 apresentamos nossas conclusões.

2. Trabalhos correlatos

Na seção anterior, fizemos um breve relato dos vários modelos de sistemas distribuídos. Para estes modelos, têm sido propostos algoritmos de consenso considerando diferentes modelos de falha.

Em [Aguilera et al. 1998] são apresentados dois algoritmos de consenso com recuperação de defeitos, utilizando detectores de defeitos, sendo que um dos algoritmos utiliza armazenamento estável, e o outro não. Estes algoritmos utilizam detectores de defeitos que, além de suspeitar do defeito de processos, constroem uma estimativa do número de vezes que cada processo falhou, classificando os processos como maus (*bad*) - processos instáveis, que falham e se recuperam com freqüência, ou que falharam permantemente, ou bons (*good*) - processos corretos, que nunca falharam, ou que após terem se recuperado de falha permanecem estáveis. Tanto processos quanto canais de comunicação são assumidos como sendo assíncronos.

Freiling, Lambertz e Cederbaum apresentam em [Freiling et al. 2008] algoritmos de

consenso para o modelo assíncrono, desenvolvidos a partir de algoritmos existentes para o modelo de falhas *crash-stop*.

O algoritmo Paxos, apresentado por Lamport em [Lamport 1998, Lamport 2001], executa sobre um sistema assíncrono dotado de um mecanismo para eleição de líder que apresente a propriedade mínima de que em algum momento de sua execução irá indicar como líder um processo que não irá falhar, o que irá garantir a terminação do protocolo. O Paxos tolera a recuperação de defeitos, desde que uma maioria dos processos esteja correta, para garantir tanto a terminação quanto o acordo uniforme.

Diferente destes algoritmos propostos para o consenso, assumimos um modelo híbrido de sistema distribuídos, no qual existem componentes síncronos e assíncronos.

3. Modelo Spa *(*Partitioned Synchronous)

Um sistema é composto por conjunto $\Pi = \{p_1, p_2, ..., p_n\}$ de processos que estão distribuídos em sítios possivelmente distintos de uma rede de computadores e por um conjunto $\chi = \{c_1, c_2, ..., c_m\}$ de canais de comunicação. Sítios computacionais formam topologias arbitrárias e processos se comunicam por meio de protocolos de transporte fim-a-fim. A comunicação fim-a-fim define canais que podem incluir várias conexões físicas no nível da rede. Portanto, um canal de comunicação c_i conectando processos p_i e p_j define uma relação do tipo "é possível se comunicar" entre p_i e p_j, ao invés de uma conexão ao nível da rede entre as máquinas que hospedam p_i e p_j. Assumimos que o sistema definido por processos e canais de comunicação forma o grafo simples e completo $DS(\Pi, \chi)$ com $(n \times (n-1))/2$ arestas. Particionamento de rede não é considerado em nosso modelo.

Um processo tem acesso a um relógio local com taxa de desvio limitado por ρ. Processos e canais de comunicação podem ser *timely* ou *untimely*. Timely/untimely é equivalente a *synchronous/asynchronous* como apresentado em [Dwork et al. 1988]. Contudo, os modelos parcialmente síncronos considerados em [Dwork et al. 1988] não consideram configurações híbridas onde alguns processos/canais são síncronos e outros assíncronos. Um processo p_i é dito *timely* se existe um limite superior (*upper-bound)* ϕ para a execução de um passo de computação por p_i. De forma análoga, um canal c_i é *timely* se existe um limite superior δ para o atrazo de transmissão de uma mensagem em c_i, e c_i conecta dois processos *timely*. Caso essas condições não se verifiquem, processos e canais são ditos *untimely* e os respectivos limites superiores são finitos, porém arbitrários.

Um canal $c_i = (p_i, p_j)$ implementa transmissão de mensagem em ambas as direções, de p_i para p_j e de p_j para p_i. δ and ϕ são parâmetros do sistema computacional subjacente, fornecidos por mecanismos adequados de sistemas operacionais e redes de tempo real. Também assumimos que os processos em Π sabem a QoS de todos os processos e canais antes da execução da aplicação.

É assumido a existência de um oráculo de *timeliness* definido pela função *QoS* que mapeia processos e canais para valores T ou U (timely ou untimely). Portanto, tal oráculo informa os processos sobre a QoS atual em termos de *timeliness* de processos e canais de comunicação. O oráculo é assumido ser preciso: a execução de *QoS(x)* no instante t retorna T/U se e somente se a QoS do elemento x (processo ou canal) no instante t for timely/untimely. Uma vez que assumimos que a QoS dos processos e canais é estática e conhecida, uma implementação trivial para o oráculo pode ser realizada durante uma fase de inicialização do

sistema: por exemplo, mantendo dois arrays binários, um para processos e outro para canais, cujo valor "1" or "0" representa timely e untimely, respectivamente.

Canais de comunicação são assumidos como confiáveis: não perdem ou alteram mensagens. Processos falham por parada silenciosa, mas podem recuperar-se (crash/recovery). Como em [Aguilera et al. 1998], assumimos que processos podem falhar e se recuperar seguidamente, apresentando um comportamento instavel. Estes processos podem manter este comportamento instável durante o tempo todo, ou a partir de algum momento no tempo se tornar permanentemente em execução, ou em crash. Um processo que não falha durante um intervalo de tempo de interesse, ou que após um tempo de instabilidade não mais falha, é considerado correto.

Sub-grafos Síncronos e Assíncronos

Dado $\Pi' \subseteq \Pi$, $\Pi' \neq \emptyset$ e $\chi' \subseteq \chi$, um sub-grafo de comunicação conectado $C(\Pi', \chi') \subseteq DS(\Pi, \chi)$ é síncrono se $\forall p_i \in \Pi'$ and $\forall c_j \in \chi'$, p_i e c_j são *timely*. Se essas condições não se verificam, $C(\Pi', \chi')$ é dito não síncrono. Utilizamos a notação Cs para denotar um sub-grafo síncrono e Ca para um sub-grafo não síncrono.

Partições Síncronas

Dado $Cs(\Pi', \chi')$, definimos partição síncrona como o maior sub-grafo $Ps(\Pi'', \chi'')$, tal que $Cs \subseteq Ps$. Em outras palavras, DS não contém $Cs'(\Pi''', \chi''') \supset Cs$ com $|\Pi'''| > |\Pi''|$.

Assumimos que existe pelo menos um processo correto em cada partição síncrona [1].

No sistema distribuido Spa, a propriedade de *strong partitioned synchrony* é necessaria para implementar detecção perfeita de defeitos como demonstrado em [Macêdo and Gorender 2009]

strong partitioned synchrony: $(\forall p_i \in \Pi)(\exists Ps \subset DS)(p_i \in Ps)$.

Observamos ainda que o fato de $Ps \subset DS$ exclui dessa especificação sistemas tipicamente síncronos com uma única partição com todos os processos do sistema.

Em *Spa*, mesmo que *strong partitioned synchrony* não possa ser satisfeita, é possível tirar proveito das partições síncronas existentes para implementar detectores parcialmente perfeitos, desde que alguma partição síncrona exista [Macêdo and Gorender 2009]. Definimos essa propriedade a seguir.

weak partitioned synchrony: o conjunto não vazio de processos que pertencem a partições síncronas é um sub-conjunto próprio de Π. Mais precisamente, assumindo que existe pelo menos uma partição síncrona Ps_x: $(\exists p_i \in \Pi)(\forall Ps_x \subset DS)(p_i \notin Ps_x)$.

No que se segue exploramos as propriedades de *strong partitioned synchrony* e *weak partitioned synchrony* sobre Spa para implementar um algoritmo de consenso onde processos podem falhar e se recuperar.

4. Consenso com Recuperação no Modelo *Spa*

Como já visto anteriormente, assumimos o modelo de falhas *crash-recovery*, no qual os processos podem falhar por colapso, e se recuperar, voltando a executar protocolos distribuídos

[1] Observe que essa hipótese é bastante plausível se consideramos *clusters* (partições síncronas) com tamanhos razoáveis - digamos, com mais de três unidades por *cluster*

em andamento (por exemplo, o consenso). Quando um processo se recupera de um defeito, retorna ao estado anterior à ocorrência do defeito, mantendo seus canais de comunicação e seu estado de sincronismo. Desta forma, o processo volta a ser membro de uma partição síncrona, se era membro desta partição antes do defeito, ou retorna como processo assíncrono. O mecanismo de recuperação de defeitos deve utilizar armazenamento estável para recuperar o estado do processo. O armazenamento é utilizado tanto para garantir a recuperação do estado geral do processo, inclusive a determinação de quais protocolos estão sendo executados, como para armazenar informação necessária à participação do processo nestes protocolos, como será visto na seção 4.3. A recuperação de um processo, efetuada por este mecanismo de recuperação, não é instantânea, e como utiliza armazenamento estavel, leva um tempo consideravel se comparada com a execução dos protocolos de detecção de defeitos e do consenso.

4.1. Adaptando os Detectores de Defeitos para uso com Recuperação

O protocolo para detecção de defeitos apresentado em [Macêdo and Gorender 2009] monitora processos membros de partições síncronas que não estejam apresentando falha, ou seja, não tenham sua identificação inserida no conjunto $faulty_i$ para cada processo $p_i \in \Pi$. Este algoritmo implementa um detector P no caso de a propriedade *strong partitioned synchrony* ser satisfeita, e um detector de defeitos xP, caso a propriedade *weak partitioned synchrony* seja satisfeita. Nesta última situação, existem processos que não são membros de partições síncronas e estes processos não são monitorados. O detector xP satisfaz as propriedades *Partially Strong Completeness* (todo processo pertencente a uma partição síncrona que falha será detectado por todo processo correto em um tempo finito) e *Partially Strong Accuracy* (se um processo pertencente a uma partição síncrona for detectado, este processo falhou). Para garantir a propriedade *Partially Strong Completeness*, o intervalo de monitoração é configurado para um valor sempre inferior ao tempo mínimo de recuperação dos processos.

Na figura 1 apresentamos uma versão modificada do algoritmo de detecção de defeitos, para considerar a recuperação de processos. Esta modificação foi inserida na tarefa $T3$ do algoritmo.

Na versão original deste algoritmo, mensagens *are-you-alive* continuam a ser enviadas a todos os processos, periodicamente, provocando a resposta destes através de mensagens *i-am-alive*. A tarefa $T3$ é executada quando uma mensagem *i-am-alive* é recebida, com o objetivo de cancelar o *timeout* definido para a recepção desta mensagem (linha 10). Nesta nova versão, quando a tarefa $T3$ é executada, além do cancelamento do *timeout*, caso a identificação do processo emissor da mensagem faça parte do conjunto $faulty_i$, assume-se que este processo falhou e apresentou uma recuperação desta falha. A identificação deste processo é retirada do conjunto $faulty_i$ (linha 11). Ao ter sua identificação retirada do conjunto $faulty_i$, o processo volta a ser monitorado. Processos que não são componentes de partições síncronas não são monitorados pelo detector xP, e podem falhar e se recuperar, voltando ao seu estado anterior, sem interferir com o detector. A identificação destes processos não é inserida no conjunto $faulty_i$ para os processos p_i.

A identificação de que processos são membros de partição síncrona e de quais não são é feita por um oráculo, como definido em [Macêdo and Gorender 2008, Macêdo and Gorender 2009]. O estado destes processos é definido no início da execução do sistema, permanecendo estável durante sua execução. Processos e canais de comunicação não

Task $T1$: **every** *monitoringInterval* **do**
(1) **for_each** $p_j,\ p_j \neq p_i$ **do**
(2) $timeout_i[p_j] \leftarrow CT_i()$ + $2\delta + \alpha$;
(3) *send* "are-you-alive?" message to p_j
(4) **end**
Task $T2$: **when** $\exists p_j : (p_j \notin faulty_i) \wedge (CT_i() > timeout_i[p_j]))$ **do**
(5) **if** $(QoS(c_{i/j}) = T)$ **then**
(6) $faulty_i \leftarrow faulty_i \cup \{p_j\}$;
(7) *send* notification (p_i, p_j) *to every* p_x *such that* $p_x \neq p_i \wedge p_x \neq p_j$
(8) **else** do nothing (wait for a remote notification)
(9) **end_if**
Task $T3$: **when** "I-am-alive" is received from p_j **do**
(10) $timeout_i[p_j] \leftarrow \infty$; /* cancels timeout */
(11) **if** $(p_j \in faulty_i)$ **then**
$faulty_i \leftarrow faulty_i - p_j$;
Task $T4$: **when** notification(p_x, p_j) is received **do**
(12) **if** $p_j \notin faulty_i$ **then**
(13) $faulty_i \leftarrow faulty_i \cup \{p_j\}$;
(14) **end_if**
Task $T5$: **when** "are-you-alive?" is received from p_j **do**
(15) *send* "I-am-alive" *to* p_j

Figure 1. Algoritmo do detector de defeitos xP para o processo p_i com recuperação.

alteram seu estado entre síncrono/assíncrono.

Como o número de partições síncronas existentes é estável, digamos k partições, e por definição [Macêdo and Gorender 2008, Macêdo and Gorender 2009], toda partição síncrona possui ao menos um processo que não falha, permanecendo correto, temos no mínimo k processos que não falham durante a sua execução.

4.2. Mecanismo para eleição de líder

Assumimos a existência de um mecanismo para eleição de líder para o modelo *Spa*, baseado no detector de defeitos utilizado (classe P ou xP). Este mecanismo sempre indica como líder um processo pertencente a alguma partição síncrona (caso a propriedade *strong partitioned synchrony* seja válida será qualquer processo). Inicialmente, o mecanismo indica como líder do grupo, para cada processo $p_i \in \Pi$, o processo de menor identificação, que seja membro de alguma partição síncrona, e que não tenha a sua identificação inserida no conjunto $faulty_i$.

Durante a execução do sistema novos processos líderes podem ser indicados pelo mecanismo, à medida em que os líderes atuais falhem. Neste caso, quando o módulo do detector de defeitos de cada processo $p_i \in \Pi$ detectar a falha do líder atual, sendo a identificação deste inserida no conjunto $faulty_i$, um novo processo líder será escolhido para substituir o que falhou. Este novo líder será um processo membro de partição síncrona que não apresente falha, e cuja identificação seja a de menor número, desde que maior do que a identificação do líder a ser substituído. Desta forma, os líderes serão sempre escolhidos em ordem crescente de suas identificações. Este dispositivo evita que processos instáveis possam falhar e se recuperar com freqüência, voltando a liderar o grupo, e gerando instabilidade na execução de protocolos.

No pior caso, como existem por definição ao menos k processos corretos, um para

cada partição síncrona, quando o de menor identificação entre estes for escolhido como líder, será obtida uma estabilidade no sistema, havendo então apenas um líder indicado por todos os processos corretos do sistema.

4.3. O Algoritmo de Consenso no Modelo Spa com Recuperação de Defeitos

Nesta seção apresentamos um algoritmo de consenso que executa sobre o modelo *Spa*, com recuperação. O consenso assume que o modelo *Spa* com uma das propriedades *Weak Partitioned Synchrony* ou *Strong Partitioned Synchrony* como válidas, e a existência de um oráculo que indica quais processos pertencem a partições síncronas e quais não pertencem (utilizado caso existam processo não síncronos). O algoritmo também assume a existência de um detector de defeitos (classe P ou xP, dependendo da propriedades que é satisfeita pelo sistema), que indica para cada processo $p_i \in \Pi$, através do conjunto $faulty_i$, que processos membros de partições síncronas falharam e permanecem com falha. Este algoritmo também utiliza um mecanismo para eleição de líder baseado no detector de defeitos utilizado, o qual é descrito na subseção 4.2. Este protocolo de consenso executa sem modificações com um detector de defeitos de qualquer uma das classes definidas.

A estrutura básica deste algoritmo é baseada no algoritmo Paxos, apresentado por Lamport em [Lamport 1998, Lamport 2001]. O consenso é realizado em 2 fases, identificadas como PREPARE-REQUEST (preparação da rodada) e ACCEPT-REQUEST (rodada para proposição e aceitação de valor).

O algoritmo é dividido em cinco tarefas, e cada processo executa dividido em três agentes: Proposer, Acceptor e Learner. A tarefa $T0$ é executada sempre que um novo processo é indicado como líder do grupo, pelo mecanismo de eleição de líder, e neste caso a tarefa $T1$ passa a ser executada pelo agente Proposer deste processo. As tarefas $T2$ e $T3$ são executadas pelos agentes Acceptor de todos os processos, e são iniciadas pela recepção das mensagens PREPARE-REQUEST e ACCEPT-REQUEST, respectivamente. A tarefa $T4$ é executada pelos agentes Learner dos processos, o tempo todo.

A tarefa $T1$ é executada pelo agente Proposer do processo líder do grupo e coordenador da rodada. Esta tarefa é dividida em duas fases: na primeira fase uma nova rodada é proposta e na segunda fase um valor é proposto para o consenso. Na $Fase1$ o Proposer propõe a nova rodada realizando um *broadcast* da mensagem PREPARE-REQUEST com o número da nova rodada proposta (linha 5 do algoritmo), e esperando por mensagens ACK-PREPARE-REQUEST de todos os processos membros de partições síncronas que não tenham falhado (linha 6). Como já foi definido anteriormente, assumimos como processo correto aquele que não falha, ou que após ter falhado e se recuperado, se mantenha em execução estável. As mensagens ACK-PREPARE-REQUEST apresentam a última rodada na qual cada processo participou (r_j), e qual foi o valor proposto nesta rodada (v_j). Na $Fase2$ um valor é escolhido entre aqueles informados nas mensagens ACK-PREPARE-REQUEST recebidas, sendo relativo à rodada mais recente na qual os processos tenham participado. É realizado um *broadcast* da mensagem ACCEPT-REQUEST, com a rodada em execução e o valor escolhido.

Function Consensus (v_i)

Task $T0$:
(1) $r_i \leftarrow 0$; $lr_i \leftarrow 0$;
(2) Every time $Leader = p_i$
(3) **Task T1** -Proposer;
Task $T1$ - Proposer:
(4) $r_i \leftarrow max(r_i, lr_i) + n$;
————————Fase 1 da rodada r: PREPARE-REQUEST————————
(5) *broadcast* PREPARE-REQUEST(r_i);
(6) **wait until** $\big($(ACK-PREPARE-REQUEST(r_i, r_j, v_j) has been received from every correct process which is a member of a synchronous partition) $\vee\ (lr_i > r_i)\big)$;
————————Fase 2 da rodada r: accept request————————
(7) **if** $\big((r_i \geq lr_i) \wedge (\neg\exists$(ACK-PREPARE-REQUEST$(r_i, r_j, v_j)$ received from p_j) such that $r_j > r_i)\big)$**then**
(8) if $\exists$ ACK-PREPARE-REQUEST(r_i, r_j, v_j) received from p_j such that $\big(\forall k$: (ACK-PREPARE-REQUEST(r_i, r_k, v_k) have been received from p_k: $r_j \geq r_k) \wedge ((r_k = r_j) \rightarrow (v_k = v_j))\big)$ then
(9) $v_i \leftarrow v_j$;
(10) *broadcast* ACCEPT-REQUEST(r_i, v_i);
Task $T2$ - Acceptor:
(11) **upon** the reception of PREPARE-REQUEST(r_c) from p_c do
(12) **if** $r_c > max(r_i, lr_i)$ **then**
(13) *send* ACK-PREPARE-REQUEST(r_c, r_i, v_i) to p_c;
(14) $lr_i \leftarrow r_c$;
Task $T3$ - Acceptor:
(15) **upon** the reception of ACCEPT-REQUEST(r_c, v_c) from p_c do
(16) **if** $r_c \geq max(lr_i, r_i)$ **then**
(17) $lr_i \leftarrow r_c$; $v_i \leftarrow v_c$; $r_i \leftarrow r_c$
(18) *broadcast* ACK-ACCEPT-REQUEST(r_i, v_i);
Task $T4$ - Learner:
(19) **wait until** ACK-ACCEPT-REQUEST(r_j, v_j) have been received from every correct process p_k which is a member of a synchronous partition, for the same round r_j or a DECISION-MESSAGE (v) is received
(20) *broadcast* DECISION-MESSAGE(v);
(21) *return* (v_j)

Figure 2. Algoritmo de consenso

A tarefa $T2$ é executada pelos agentes Acceptors de todos os processos. Um processo inicia a execução desta tarefa ao receber uma mensagem PREPARE-REQUEST de um Proposer, sendo que a rodada proposta deve ser mais recente do que qualquer rodada na qual o processo esteja participando ou tenha participado (verificado pelo if da linha 12). Nesta tarefa o processo informa o seu valor atual e a última rodada na qual participou, através da mensagem ACK-PREPARE-REQUEST, enviada ao Proposer.

A tarefa $T3$ é executada pelos agentes Acceptors de todos os processos. Um processo inicia a execução desta tarefa ao receber uma mensagem ACCEPT-REQUEST de um Proposer, sendo que a rodada referenciada na mensagem deve ser mais recente do que qualquer outra rodada na qual o processo esteja participando ou tenha participado. O processo atualiza o seu valor proposto e o seu número de rodada, e realiza um broadcast com mensagem ACK-ACCEPT-REQUEST, com o número de rodada atual, e o valor proposto.

A tarefa $T4$ é executada o tempo todo pelos agentes Learner de todos os processos, recebendo as mensagens ACK-ACCEPT-REQUEST enviadas pelos agentes Acceptor dos processos. Quando um Learner receber mensagens para uma mesma rodada de um Quorum de todos os processos membros de partição síncrona que não estejam apresentando falha, ele termina o consenso e indica o valor adotado nesta rodada como o valor acordado.

Em dois momentos da execução do algoritmo se espera por mensagens de um quorum de processos: - na linha 6 o coordenador da rodada espera por mensagens ACK-PREPARE-REQUEST de todos os processos membros de partições síncronas que não estão com falha; - na linha 19 todos os processos executam um wait, no qual esperam por mensagens ACK-ACCEPT-REQUEST de uma mesma rodada, que tenham sido recebidas também por um quorum de processos formado de forma similar ao da linha 6.

Este algoritmo necessita de armazenamento estável, para garantir que um agente Acceptor não execute a tarefa $T4$ para uma rodada, já tendo executado a tarefa $T3$ para uma rodada de número superior. Este dispositivo garante o compromisso dos processos de não participarem de rodadas de número inferior ao número da última rodada na qual já participaram [Lamport 1998, Lamport 2001].

Provas Formais Nas sub-seções que se seguem apresentamos as provas formais para as propriedades do consenso: Terminação, Acordo Uniforme e Validade.

4.3.1. Terminação

O protocolo de consenso apresentado satisfaz a propriedade terminação, sendo a sua prova apresentada no Teorema 1, a seguir.

Theorem 1. *Assumimos a existência do modelo Spa, que o sistema é equipado com um detector de defeitos P ou xP e o mecanismo para eleição de líder descrito na seção 4.2. Todo processo correto decide.*

Proof

A prova é desenvolvida por contradição, assumindo que o consenso nunca é obtido. Isto seria possível se o consenso ficasse bloqueado em algum dos comandos wait executados, nas linhas 6 e 19 do algoritmo, ou se o algoritmo executasse eternamente sendo que a cada rodada iniciada o mecanismo de eleição de líder detectaria a falha do líder atual, o qual é o coordenador da rodada atual, e passaria a indicar um novo processo líder, que iniciaria uma nova rodada, não permitindo o término da rodada atual e a obtenção do consenso.

- Primeiramente verificamos a possibilidade de o algoritmo bloquear no comando wait da linha 6 ou 19.
  - O algoritmo não bloqueia ao executar o comando wait da linha 6, ao esperar por mensagens de todos os processos corretos que são membros de partições síncronas, uma vez que, todo processo membro de partição síncrona que falha será detectado pelo detector de defeitos (propriedade *Partially Strong Completeness* ou *Strong Completeness*, dependendo da classe do detector).
  - O algoritmo não bloqueia ao executar o comando wait da linha 19 devido à mesma argumentação do ítem anterior, aplicada às mensagens de cada rodada, e

considerando que este wait espera por mensagens de diversas rodadas de forma concorrente, até que ocorra uma rodada com decisão.

- Possibilidade de o algoritmo executar eternamente sem obter o consenso:
  - Como existem ao menos k processos corretos, que não irão falhar em nenhum momento, nas k partições síncronas, e como o mecanismo para eleição de líder só escolhe processos membros de partições síncronas como líder, e em ordem crescente, no momento em que este mecanismo escolher um destes k processos como líder, após o início de uma nova rodada por este processo esta rodada será terminada e o consenso será obtido.

Portanto, como o algoritmo não bloqueia, e não há a possibilidade de as rodadas serem sempre executadas sem encerrarem, o consenso termina a sua execução, o que contradiz a suposição inicial da prova.

$\square_{Theorem\ 1}$

4.3.2. Validade

Todos os processos iniciam com algum valor, e apenas valores indicados pelos processos podem ser utilizados para o acordo.

Theorem 2. *Se um processo decide, o faz pelo valor inicial de algum dos processos participantes do consenso.*

A prova deste teorema se baseia fato de que no início os processos só propõem seu valor inicial, e que a cada rodada, os processos propõem seu valor inicial, ou o valor inicial de algum outro processo, adquirido em uma rodada anterior. Esta prova não será apresentada neste trabalho.

4.3.3. Acordo Uniforme

O consenso apresentado na Figura 2 satifaz a propriedade consenso uniforme. A prova para esta propriedade está descrita no Teorema 3, a seguir.

Theorem 3. *Se um processo decide por um valor v, todos os processos que decidem o fazem pelo mesmo valor v.*

Proof A prova deste teorema é desenvolvida por contradição, assumindo que dois processos, p_x e p_y decidem pelos valores v_x e v_y, respectivamente, e que $v_x \neq v_y$.

Vamos considerar duas possibilidades:

1. Os processos p_x e p_y decidem na mesma rodada:
   - Se os processos p_x e p_y decidem na mesma rodada, o fazem pelo valor recebido nas mensagens ACK-ACCEPT-REQUEST, enviadas por um Quorum de processos. O valor encaminhado nestas mensagens é valor v_c, proposto pelo processo p_c da rodada, sendo, portanto o mesmo em todas as mensagens. Portanto, neste caso p_x e p_y decidem pelo mesmo valor, e $v_x = v_y$.
2. Os processos p_x e p_y decidem em rodadas diferentes:

- Assumimos que o processo p_x decide pelo valor v_x na rodada r_x, e que p_y decide pelo valor v_y, na rodada r_y, posterior. p_x decide ao executar a tarefa $T4$ do algoritmo, e receber mensagens ACK-ACCEPT-REQUEST para esta rodada de todos os processos membros de partição síncrona que não falharam (linha 19 do algoritmo). Chamaremos este grupo de processos de q_x. Todos estes processos assumiram como seu valor proposto (v_i) o valor proposto pelo coordenador da rodada na mensagem ACCEPT-REQUEST, registrando a rodada na qual o valor foi recebido (linha 17). O Processo p_y decide pelo valor v_y em uma rodada r_y posterior, ao receber mensagens ACK-ACCEPT-REQUEST com este valor de todos os processos membros de partição síncrona que não apresentem falha, executando a tarefa $T4$. Estas mensagens foram enviadas pelos processos ao receber a mensagem ACCEPT-REQUEST com este mesmo valor, do coordenador da rodada, processo p_{cy} (tarefa $T3$). O coordenador da rodada, processo p_{cy} escolhe o valor proposto entre os recebidos em mensagens ACK-PREPARE-REQUEST de todos os processos membros de partição síncrona que não estão com falha (linhas 6 a 9). Iremos chamar este grupo de processos de q_{cy}. Estas mensagens são enviadas pelos processos ao executar a tarefa $T2$, em resposta à mensagem PREPARE-REQUEST do coordenador. O coordenador escolhe o valor relativo à rodada mais recente. Como temos que no mínimo k processos são corretos e nunca falham (assumido pelo modelo), estes k processos são membros tanto do conjunto q_x quanto do conjunto q_{cy}, portanto $q_x \cap q_{cy} \neq \oslash$. Temos que, qualquer que seja a rodada seguinte à rodada r_x, estes k processos participarão desta rodada, e o valor a ser assumido pelo coordenador desta nova rodada será o mesmo valor assumido por estes processos na rodada r_x, ou seja, v_x. Portanto, $v_y = v_x$.

Consequentemente, em qualquer possibilidade de execução p_x e p_y decidem pelo mesmo valor, e $v_x = v_y$. $\square_{Theorem\ 3}$

5. Conclusão

Apresentamos neste artigo um novo algoritmo de consenso, desenvolvido para executar sobre o modelo Spa, que tolera a recuperação de defeitos de processos. Este consenso executa utilizando um detector de defeitos, que pode ser da classe P, caso a propriedade *Strong Partitioned Synchrony* seja satisfeita, ou da classe xP, caso a propriedade válida seja a *Weak Partitioned Synchrony*. O algoritmo é baseado no Paxos, apresentado por Lamport em [Lamport 1998].

Este algoritmo não apresenta restrição no número de defeitos e de recuperações de defeitos, apresentadas pelos processos, uma vez que tanto a terminação como o acordo são garantidos pela existência de ao menos k processos corretos, um em cada partição síncrona do sistema. Diferente de outros algoritmos de consenso para modelos parcialmente síncronos, não existe restrição imposta pelo consenso no número de processos corretos para que o consenso seja obtido.

O algoritmo proposto utiliza um mecanismo simples para eleição de líder, o qual garante a terminação do consenso, ao garantir que líderes são escolhidos na ordem crescente de seus identificadores. Este mecanismo é baseado no detector de defeitos sendo utilizado.

References

Aguilera, M. K., Chen, W., and Toueg, S. (1998). Failure detection and consensus in the crash-recovery model. In *DISC '98: Proceedings of the 12th International Symposium on Distributed Computing*, pages 231–245.

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

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

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

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

Freiling, F. C., Lambertz, C., and Majster-Cederbaum, M. (2008). Easy consensus algorithms for the crash-recovery model. In *DISC '08: Proceedings of the 22nd international symposium on Distributed Computing*, pages 507–508, Berlin, Heidelberg. Springer-Verlag.

Gorender, S. and Macêdo, R. J. A. (2002). Um modelo para tolerância a falhas em sistemas distribuídos com qos. In *Anais do Simpósio Brasileiro de Redes de Computadores, SBRC 2002*, pages 277–292.

Gorender, S., Macêdo, R. J. A., and Raynal, M. (2007). An adaptive programming model for fault-tolerant distributed computing. *IEEE Transactions on Dependable and Secure Computing*, 4(1):18–31.

Lamport, L. (1998). The part time parliament. *ACM Trans. on Computer Systems*, 16(2):133–169.

Lamport, L. (2001). Paxos made simple. *Distributed Computing Column of ACM SIGACT News*, 32(4):51–58.

Lynch, N. A. (1996). *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc.

Macêdo, R. J. A. (2007). An integrated group communication infraestructure for hybrid real-time distributed systems. In *9th Workshop on Real-Time Systems*, pages 81–88.

Macêdo, R. J. A. and Freitas, A. (2009). A generic group communication approach for hybrid distributed systems. (5523(4)):102–115.

Macêdo, R. J. A. and Gorender, S. (2008). Detectores perfeitos em sistemas distribuídos não síncronos. In *IX Workshop de Teste e Tolerância a Falhas (WTF 2008), Rio de Janeiro, Brazil*.

Macêdo, R. J. A. and Gorender, S. (2009). Perfect failure detection in the partitioned synchronous distributed system model. In *Proceedings of the The Fourth International Conference on Availability, Reliability and Security (ARES 2009), IEEE CS Press*. To appear in an extended version in Int. Journal of Critical Computer-Based Systems (IJCCBS).

Macêdo, R. J. A., Gorender, S., and Raynal, M. (2005). A qos-based adaptive model for fault-tolerant distributed computing (with an application to consensus). In *Proceedings of IEEE/IFIP Int. Conference on Computer Systems and Networks (DNS05)*, pages 412–421.

Veríssimo, P. and Casimiro, A. (2002). The timely computing base model and architecture. *IEEE Transactions on Computers*, 51(8):916–930.

Garantindo a Circulação e Unicidade do Token em Algoritmos com Nós Organizados em Anel Sujeitos a Falhas *

Luciana Arantes[1], Julien Sopena[1]

[1] LIP6 - Université de Paris 6 - INRIA Rocquencourt
Paris, France.

Luciana.Arantes@lip6.fr, Julien.Sopena@lip6.fr

Abstract. *We present in this paper an algorithm that provides some few functions which, when called by existing token ring-based distributed algorithms, easily render its token fault tolerant to crashes. It ensures thus the circulation and uniqueness of the token in presence of node crashes. At most k consecutive node crashes are tolerated. The lost of the token is avoided by keeping temporal backup copies of it in k other nodes. Our algorithm scales very well since a node monitors the liveness of at most k other nodes and neither a global election algorithm nor broadcast primitives are used to regenerate a new token. Its token regeneration mechanism is very effective in terms of latency cost. Furthermore, if the token keeps some information, the latter can be easier restored in case of failure of the token holder.*

Resumo. *Apresentamos neste artigo um algoritmo que oferece um conjunto pequeno de funções que, quando chamadas por algoritmos distribuídos do tipo token-ring, fazem com que o token utilizado por estes se torne tolerante a falhas. Ele garante assim a circulação e unicidade to token em presença de falhas por parada dos nós. O número máximo de falhas consecutivas toleradas é limitado a k. A perda do token é evitada mantendo-se cópias temporárias deste em k outros nós. O algoritmo é escalável pois um nó monitora no máximo k nós, não necessita de uma eleição de líder, que envolva todos os nós do sistema e nem a utilização de primitivas de difusão para recriar um novo token. Nossa solução para recriá-lo é bastante eficaz em termos de latência. Além disso, se o token armazena alguma informação, esta pode ser mais facilmente restaurada em caso de falha do nó que detém o token.*

1. Introdução

Inúmeros algoritmos distribuídos baseados em token-ring (exclusão mútua (*mutual exclusion*) [Lann 1977], detecção do término de uma aplicação distribuída (*termination detection*) [Misra 1983] [Dijkstra et al. 1986], eleição do líder (*leader election*) [Chang and Roberts 1979], [Franklin 1982], [Peterson 1982], algoritmos para ordenar a difusão de mensagens (*total order broadcast)* [Défago et al. 2004], gestão de filiação de grupo (*group membership*) [Amir et al. 1995], etc.) são baseados na unicidade de um token que circula ao longo de todos os nós do sistema, organizados logicamente em um anel. A todo momento existe no máximo um nó que possui o token (propriedade de segurança - *safety*) e o seu detentor sempre o envia ao seu nó sucessor no anel lógico, ou seja, o token circula entre todos os nós (propriedade de vivacidade - *liveness*).

*Mais valem $k + 1$ tokens voando do que um na mão.

Entretanto, em caso de falha de um nó, os mecanismos para detecção da perda do token e a sua regeneração podem ser caros e não muito eficazes, principalmente se considerarmos o aspecto escalabilidade, ou seja, o número de nós do anel (larga escala). Soluções existentes geralmente consistem em periodicamente verificar se a configuração do anel mudou (falha de um ou mais nós) através do monitoramento de todos os nós. Se houver alguma mudança, é necessário então executar um algoritmo de eleição global a fim de regerar um novo token e eleger o seu detentor (*token holder*). Em termos de escalabilidade, o ideal seria não envolver todos os nós do sistema para eleger um novo detentor do token (*token holder*) em caso de falha do anterior, não monitorar todos os nós do sistema para detectar falhas de nós e nem utilizar primitivas de difusão (*broadcast*). Por razões de desempenho, a solução deve também evitar reconstruir o anel lógico. Um terceiro ponto importante é que se o token armazenar alguma informação como em [Misra 1983] [Défago et al. 2004], esta deve ser facilmente restaurada quando o novo token é recriado.

Assim, com o objetivo de minimizar os problemas de falta de escalabilidade, desempenho e perda de informação do token acima relacionados, propomos um novo algoritmo que facilmente torna o token tolerante a falhas e que pode ser "plugado" em algoritmos token-ring existentes como os mencionados no início deste artigo. Para tanto, nosso algoritmo oferece as seguintes três funções: $SafeSendToken$, $SafeReceiveToken$ e $UpdateToken$. A perda do token, a criação de um novo token e a detecção de falhas dos nós se tornam assim transparentes à aplicação.

Basicamente, nossa solução evita a perda do token devido à falha de nós, criando cópias temporárias do token em outros nós que não aquele que detém o token. Sempre que o nó S_i, detentor do token, envia uma mensagem $\langle$*TOKEN*$\rangle$ ao seu sucessor direto S_{i+1} a fim de lhe passar o token, S_i também envia, de forma transparente para a aplicação, uma cópia desta mensagem aos k nós que sucedem S_{i+1} no anel. Ao receber uma mensagem $\langle$*TOKEN*$\rangle$, o nó em questão começa a monitorar um subconjunto de nós que receberam também a mesma cópia da mensagem. Um nó tem o direito exclusivo de utilizar o token ou quando ele recebe uma mensagem $\langle$*TOKEN*$\rangle$ que informa que ele é o próximo detentor do token, ou quando o mecanismo de monitoramento deste nó detecta que todos os nós que ele monitora falharam.

Nosso algoritmo tolera no máximo k falhas consecutivas em relação à ordem dos nós no anel. Os pontos críticos levantados anteriormente são evitados em nossa solução: ela é escalável pois um nó monitora no máximo k nós e o monitoramento inicia (resp. encerra) quando o token está (resp. não está) na vizinhança do nó; o token não é regerado com um algoritmo de eleição que envolva todos os nós; não é necessário reconstruir logicamente o anel; como k nós recebem um cópia do token, a informação que ele armazena pode ser mais facilmente recuperada se o nó detentor do token falhar.

Vale ressaltar que nossa estratégia para regerar o token é bastante eficaz em termos de latência quando comparada com outras em que a detecção da perda do token envolve todos os nós do anel e que utilizam uma eleição global ou protocolo de gestão de filiação de grupo para a escolha do novo detentor do token. Na nossa solução, o token é regerado instantaneamente graça às cópias backup do token. Quando a falha de um nó é detectada, o nó com o direito de recriar o token não precisa se comunicar com nenhum outro para fazê-lo. Uma segunda observação importante é que, conjuntamente com um protocolo que mantém a circulação virtual do token sobre um grafo (e.g. *depth-first token circulation protocol*), nosso algoritmo pode ser utilizado em qualquer grafo que repre-

sente uma dada topologia. Além disso, em caso de falhas (no máximo k consecutivas no anel lógico correspondente), o grafo não precisa ser reconstruído. Um último ponto é que nosso algoritmo pode suportar mais de k falhas, ou seja, tantas falhas enquanto estas não constituírem um conjunto de k falhas consecutivas.

Pode-se argumentar que nossa solução gera k mensagens adicionais para cada envio de uma mensagem $\langle \textit{TOKEN} \rangle$. Entretanto, ela conserva a mesma ordem de complexidade de mensagens $\mathcal{O}(N)$ que o algoritmo original. Além do que, nosso mecanismo de detecção de falhas apresenta um custo muito menor em termos de mensagens quando comparado com a maioria das soluções existentes em que cada nó monitora todos os outros, principalmente em sistemas com um número grande de nós.

O restante do artigo está organizado da seguinte maneira. A seção 2 especifica o modelo computacional. Nosso algoritmo, que oferece funções que permitem ao token se tornar resiliente às falhas, se encontra descrito na seção 3. Esta também inclui um esboço da prova de correção e exemplos de como algoritmos baseados em token circulando em anel podem facilmente utilizar as funções oferecidas pelo nosso algoritmo a fim de evitar a perda do token em caso de falhas. Uma comparação com trabalhos relacionados é feita em 4. Finalmente, a seção 5 conclui o trabalho.

2. Modelo de Sistema

Consideramos um sistema distribuído formado por um conjunto finito Π de $N > 1$ nós, $\Pi = \{S_0, S_2, \ldots, S_N - 1\}$ que se comunicam apenas por mensagens. Os canais são considerados confiáveis (*reliable*) e bidirecionais; eles não alteram, não criam, nem perdem mensagens. Entretanto, mensagens podem ser entregues fora da ordem de suas respectivas emissões. Há um processo por nó; os termos nós e processos são análogos neste artigo.

Os N nós são organizados em um anel lógico. Todo nó S_i é identificado de maneira única e conhece a identificação. S_i se comunica com seus respectivos $k + 1$ sucessores e predecessores mais próximos. Para evitar complicar a notação, nós denominamos que o sucessor de S_i é S_{i+1} e não $S_{(i+1)\%N}$.

Inicialmente, um certo nó possui o token. Este circula numa dada direção. Denominamos S_t o nó corrente detentor do token e este sempre o libera ao seu sucessor direto dentro de um limite de tempo.

Um processo pode falhar por parada (*crash*). Um processo *correto* é um processo que não falha durante a execução do sistema; senão ele é considerado *falho*. Seja k ($k < N - 1$), valor conhecido por todos os processos, o número máximo de falhas consecutivas toleradas no anel.

O sistema é síncrono, o que significa que a velocidade relativa dos processadores e os atrasos nas entregas das mensagens pelos canais de comunicação atendem a limites estabelecidos e conhecidos. Esta hipótese é necessária para garantir a unicidade to token, ou seja, um processo não pode ser suspeitado erroneamente de se encontrar falho.

3. Algoritmo em Anel Tolerante a Falhas do Token

Apresentamos nesta seção nosso algoritmo que torna o token, que circula por processos organizados logicamente em um anel, tolerante a falhas. Ele oferece três funções

à aplicação: $SafeSendToken$, $SafeReceiveToken$ e $UpdateToken$. A aplicação deve então substituir as suas funções originais utilizadas para enviar o token ao nó sucessor no anel e receber o token do predecessor pelas funções $SafeSendToken$ e $SafeReceiveToken$ respectivamente. A função $UpdateToken$ pode ser utilizada pela aplicação quando, em caso de falha, as informações guardadas pelo token precisarem ser atualizadas (veja seção 3.4). Consideramos que a aplicação se comporta corretamente, ou seja, um nó utiliza o token pour um intervalo de tempo limitado e depois o envia ao seu sucessor direto no anel. Consideramos que inicialmente a aplicação sempre atribui o token a S_0.

Tanto a versão original do algoritmo da aplicação quanto a sua versão que utiliza as funções oferecidas por nosso algoritmo para tolerar falhas do token precisam assegurar as seguintes propriedades:

- **segurança** (*safety*): A todo instante, existe, no máximo, um token no sistema.
- **vivacidade** (*liveness*): A todo instante, todo processo correto receberá o token num intervalo de tempo limitado.

Note que, momentaneamente, pode acontecer que não exista nenhum token no sistema pois ele está sendo regerado. No caso do nosso algoritmo, as cópias do token não são consideradas como token enquanto não substituírem o verdadeiro.

3.1. Variáveis e Mensagens

O nó S_i possui as variáveis locais $count_{S_i}$ e $token_{S_i}$. Aquela é utilizada para evitar que um nó considere uma mensagem $\langle$*TOKEN*$\rangle$ antiga como válida enquanto que esta controla se S_i detém o token, uma cópia deste, ou nenhum dos dois. Os valores $REAL$, $BACKUP$ e $NONE$ podem ser atribuídos à variável $token_{S_i}$: (1) $token_{S_i}$ possui o valor $REAL$ sempre que S_i tiver o direito de utilizar o token, ou seja, o processo da aplicação que executa em S_i pode usá-lo. Num dado instante t, apenas um processo tem sua variável $token$ igual a $REAL$, o que assegura a unicidade do token; (2) o valor $BACKUP$ é atribuído a $token_{S_i}$ se S_i é um dos k sucessores diretos do nó S_t e detém uma cópia válida do token. Se S_i não falhar, haverá um momento em que o valor de $token_{S_i}$ será substituído por $REAL$; (3) O valor $NONE$ é atribuído à variável $token_{S_i}$ quando S_i não possui o token nem uma cópia dele.

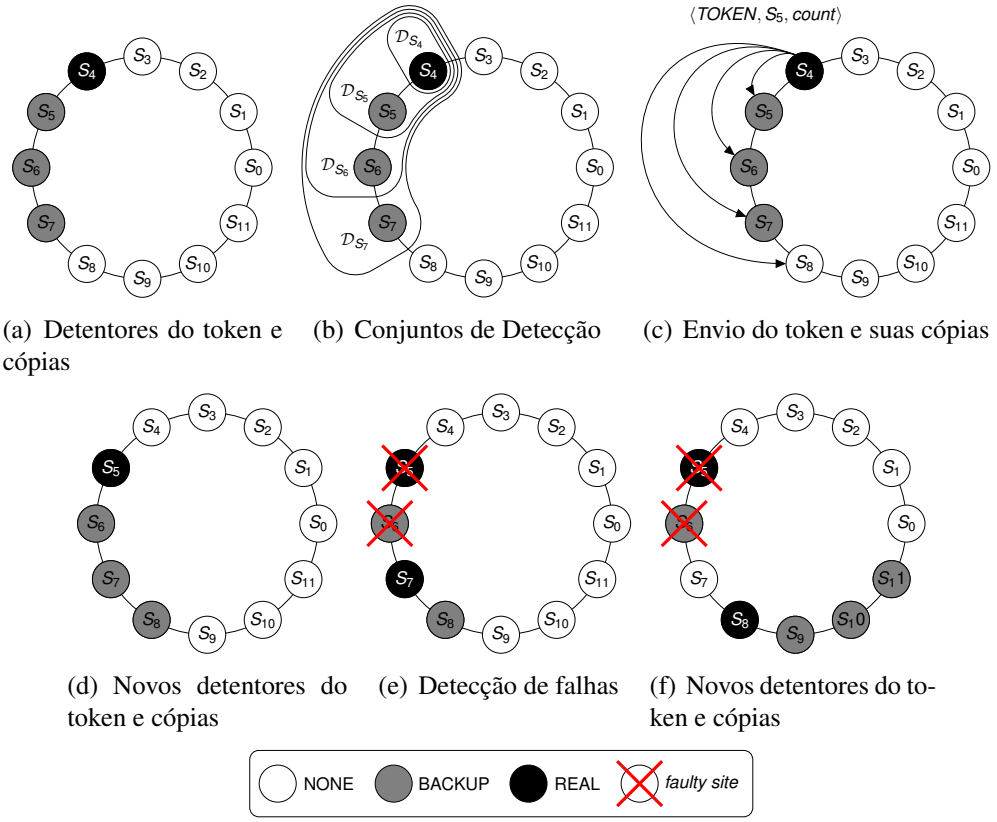
Os seguintes dois conjuntos são utilizados por S_i:

- $\mathcal{D}_{S_i}$ (Conjunto de detecção, *Detection set*): Conjunto que inclui os nós que S_i precisa monitorar para detectar as respectivas falhas além do próprio S_i. Ele é composto por $\{S_t \ldots S_i\}$, ou seja, o conjunto de nós entre S_t e S_i na ordem crescente do anel, incluindo ambos os nós. Ele possui então no máximo $k+1$ nós.
- $\mathcal{F}_{S_i}$ (Conjunto de nós falhos, *Faulty set*): Conjunto de nós que S_i detectou como falhos.

Se o valor de $token_{S_i}$ for igual a $REAL$ ou $BACKUP$, então $\mathcal{D}_{S_i} \neq \emptyset$. Os $k+1$ conjuntos de detecção são construídos de forma que cada um difere do outro e um inclui o outro (*nested detection sets*). O nó que detém o token está presente em todos os $k+1$ conjuntos. A vantagem de tal construção é o baixo custo em termos de mensagens quando comparado a um sistema de monitoramento no qual cada um dos $k+1$ nós monitora os outros k. Além disso, em caso de falha, a eleição do novo detentor do token não requer nenhum envio de mensagem.

A Figura 1 ilustra $N = 12$ nós organizados em anel e $k = 3$. Na Figura 1(a), o nó S_4 é o detentor do token e os nós $S_5 \ldots S_7$ possuem cópias do token enquanto que na Figura 1(b), podemos observar o conjunto de detecção dos nós $S_4 \ldots S_7$.

A mensagem $\langle TOKEN \rangle$ contém a identificação do sucessor do nó emissor da mensagem além do valor corrente da variável $count$ deste nó. O token pode conter outros dados relacionados ao algoritmo que utiliza nossas funções.



(a) Detentores do token e cópias (b) Conjuntos de Detecção (c) Envio do token e suas cópias

(d) Novos detentores do token e cópias (e) Detecção de falhas (f) Novos detentores do token e cópias

Figura 1. Exemplo execução algoritmo tolerante a falhas do token com k=3

3.2. Algoritmo

A Figura 2 descreve o pseudo-código para o nó S_i do nosso algoritmo. Por razões de simplificação, o monitoramento e a detecção da falha dos nós que pertencem a $\mathcal{D}_{S_i}$ não foram incluídos na figura. Sempre que $\mathcal{D}_{S_i}$ é atualizado, a função $updateDectection(\mathcal{D}_{S_i})$ desativa o monitoramento dos processos do antigo conjunto $\mathcal{D}_{S_i}$ e ativa o monitoramento dos processos contidos no novo $\mathcal{D}_{S_i}$. Quando a falha de S_j é detectada, S_i recebe o evento *Suspected* (função $ReceiveSuspected$). Como o sistema é síncrono não há nunca falsas suspeitas de falhas. Uma segunda observação é que S_i pertence ao seu próprio conjunto $\mathcal{D}_{S_i}$ não para se auto-monitorar mas apenas para decidir mais facilmente que ele detém o $REAL$ token quando todos os nós que ele monitora falharem. Um terceiro ponto a ressaltar é que em termos de implementação, não é necessário que S_i monitore todos os de seu $\mathcal{D}_{S_i}$ ao mesmo tempo. Ele pode monitorar apenas o predecessor não falho mais perto dele no anel. Se este também falhar, ele então passa a monitorar o predecessor deste nó, que pertence ao seu conjunto $\mathcal{D}_{S_i}$ e assim sucessivamente.

Na fase de inicialização (linhas 3-15), consideramos que S_0 detém o token. Consequentemente, os nós S_1 a S_k possuem uma cópia do token (token = $BACKUP$) e seus respectivos $\mathcal{D}$s são inicializados a $\{S_0, \ldots S_i\}$. Cada um destes nós começa então, com exceção dele mesmo, a monitorar os processos contidos no seu $\mathcal{D}$ (linha 15). Todos os outros nós do anel não possuem nenhuma cópia válida do token ($token = NONE$).

A função $SafeSendToken(< TOKEN >)$ permite a $S_i = S_t$ enviar o token a seu sucessor S_{i+1} e uma cópia deste a $\{S_{i+2} \ldots S_{t+k+1}\}$, indicando que o próximo proprietário do $REAL$ token é o nó S_{i+1}, i.e. $S_{i+1} = S_t$ (Figuras 1(c) e 1(d)). Como S_i não possui mais o token ($token = NONE$), ele não mais monitora os processos de seu $\mathcal{D}_{S_i}$ (linha 21). Vale lembrar que consideramos que S_i chama a função $SafeSendToken(< TOKEN >)$ somente se ele detiver o $REAL$ token e ao fazê-lo ele não pode mais utilizá-lo até adquiri-lo novamente.

Ao receber uma mensagem $TOKEN$ de S_j (função $SafeReceiveToken(< TOKEN, S_t, count_r >)$) que não seja antiga, S_i atualiza sua variável $count_{S_i}$ com o valor ($count_r$) contido na mensagem $TOKEN$ (linha 24). Ele então atribui ao seu $\mathcal{D}_{S_i}$ os nós entre S_t e ele próprio, incluindo ambos os nós (linha 25). Assim, (1) se S_i é o próximo detentor do token, ele atribui o valor $REAL$ à sua variável $token_{S_i}$ e entrega o token à aplicação (linhas 27-28); (2) Se S_i detecta a falha de todos os nós que ele monitora, ele se torna o detentor do $REAL$ token executando para tanto a função $UseBackup()$ (linhas 39-42; Figura 1(e)). Ele entrega da mesma forma que em (1) o token à aplicação; (3) senão S_i afeta o valor $BACKUP$ à sua variável $token_{S_i}$ (linha 32). Nos três casos, ele passa a monitorar os nós do seu novo $\mathcal{D}_{S_i}$ (linha 33).

Quando S_i é informado ($ReceiveSuspected(S_j)$, linha 34) da falha de S_j, um dos nós monitorado por S_i, este atualiza o conteúdo de seu conjunto de nós falhos $\mathcal{F}_{S_i}$. Se S_i detectar a falha de todos os nós que ele monitora, S_i chama a função $UseBackup()$, a fim de se tornar o novo detentor do $REAL$ token.

A função $UseBackup()$ adiciona a $count_{S_i}$ o número de nós que S_i detectou terem falhados (linha 39), o que garante a coerência de $count_{S_i}$ e que mensagens antigas que possam chegar mais tarde a S_i sejam descartadas. Ela também altera o valor da variável $token_{S_i}$ de $BACKUP$ para $REAL$ (linha 40). Antes de entregar o token à aplicação, S_i pode atualizar a informação armazenada no token (linha 41), se necessário. Por exemplo, o algoritmo de detecção de término da aplicação de Misra [Misra 1983] guarda um contador no token e este precisa ser atualizado antes que o token seja entregue à aplicação se o novo detentor do token não o recebeu de seu nó predecessor. Para mais detalhes veja a seção 3.4. Vale ressaltar que a criação de um novo token é bastante eficaz pois basicamente consiste em considerar uma das cópias $BACKUP$ do token como o $REAL$ token.

3.3. Esboço da Prova de Correção

Alguns dos principais argumentos da prova de que o algoritmo da Figura 2 satisfaz as propriedades de segurança (*safety*) e vivacidade (*liveness*) são apresentados nesta subseção.

Definições: Consideramos que o tempo é discretizado pela chamada às funções $SafeSendToken$, $SafeReceiveToken$ e $UseBackup$. $t = 0$ é estabelecido pela chamada da função $Initialisation$. $\mathcal{C}$ denota o tempo discretizado enquanto que $\mathcal{C}_d$, o tempo discretizado referente às chamadas das funções $UseBackup$. Processos não têm

```
1  /* 𝒟: Detection set */
2  /* ℱ: Faulty set    */
3  Initialisation ()
4      count ← 0
5      ℱ ← { }
6      case i = 0
7          token ← REAL
8          𝒟 ← {S_0}
9      case 0 < i ≤ k
10         token ← BACKUP
11         𝒟 ← {S_0, .., S_i}
12     case k < i
13         token ← NONE
14         𝒟 ← { }
15     UpdateDetection(𝒟)

16 SafeSendToken (⟨TOKEN⟩)
   to S_{i+1}
17     count ← count + 1
18     Send ⟨TOKEN,S_{i+1},count ⟩
       to   {S_{i+1}, .., S_{i+k+1}}
19     token ← NONE
20     𝒟 ← { }
21     UpdateDetection(𝒟)

22 SafeReceiveToken (⟨TOKEN,S_t,count_r⟩) from S_j
23     if count < count_r then
24         count ← count_r
25         𝒟 ← {S_t, .., S_i}
26         if S_i = S_t then
27             token ← REAL
28             DeliverToken(⟨TOKEN⟩)
29         else if 𝒟/ℱ = {S_i} then
30             UseBackup()
31         else
32             token ← BACKUP
33         updateDetection(𝒟)

34 ReceiveSuspected (S_j)
35     ℱ ← ℱ ∪ S_j
36     if 𝒟/ℱ = {S_i} then
37         UseBackup()

38 UseBackup ()
39     count ← count + ( #(𝒟) - 1 )
40     token ← REAL
41     UpdateToken(⟨TOKEN⟩)
42     DeliverToken(⟨TOKEN⟩)
```

Figura 2. Algoritmo tolerante à perda do token

acesso a $\mathcal{C}$ e nem a $\mathcal{C}_d$. Eles são introduzidos apenas por uma questão de conveniência da apresentação da prova.

Para expressar o valor de certas variáveis de nosso algoritmo em relação a $\mathcal{C}$, definimos algumas funções. Para um dado $t \in \mathcal{C}$ e um dado site S, cada uma destas funções retorna o valor da respectiva variável.

Denotamos $\mathcal{P}(\Pi)$ o conjunto de potência de π. As funções que respectivamente retornam o valor das variáveis *token*, $count$ e $\mathcal{D}$ para o nó S a t são:

$Token(S,t):\ \pi \times \mathcal{C} \rightarrow \{NONE, BACKUP, REAL\}$
$Count(S,t):\ \pi \times \mathcal{C} \rightarrow \mathbb{N}$
$\mathcal{D}(S,t):\ \pi \times \mathcal{C} \rightarrow \mathcal{P}(\Pi)$

$\langle Pend_{REAL} \rangle$ e $\langle Pend_{BACKUP} \rangle$ denotam uma mensagem $\langle$*TOKEN,*S_i*,count*$\rangle$ pendente que é endereçada respectivamente a S_i e um dos k sucessores de S_i.

Denotamos S_d o nó que chama a função $UseBackup$.

Para auxiliar a prova de nosso algoritmo, introduzimos as seguintes propriedades:

- $\boldsymbol{PSafetyCond(t)}$: Existe no máximo um processo correto que detém o $REAL$ token ou que é o receptor de $\langle Pend_{REAL} \rangle$.
  Se $PSafetyCond(t)$ é satisfeita, nós denotamos:
  - $Holder(t)$: o nó que satisfaz $PSafetyCond(t)$.
  - $HCount(t)$: equivale a $Count(Holder(t), t)$, se $Token(Holder(t), t) = REAL$ ou ao valor de *count* de $\langle Pend_{REAL} \rangle$, caso contrário.
  - $\mathcal{T}(t)$: o conjunto token a t, ou seja, o conjunto ordenado de $k+1$ nós composto de $Holder(t)$ e seus k sucessores.
- $\boldsymbol{PHolderCount(t)}$: O detentor do token possui o maior valor de *count* entre todos os nós não falhos e mensagens $\langle$*TOKEN*$\rangle$ pendentes.
- $\boldsymbol{PHolderMonitored(t)}$: Se um nó não falho monitora outros nós, então $Holder(t)$ está presente entre estes nós.
- $\boldsymbol{PNestedMonitoring(t)}$: Se dois nós não falhos monitoram outros nós, ao menos um deles monitora o outro.

Definimos que $S_i \prec_t S_j$, se S_i precede S_j em $\mathcal{T}(t)$.

Assumimos que o algoritmo que utiliza as funções oferecidas por nosso algoritmo as chamam corretamente e que o algoritmo original (sem as chamadas às referidas funções) satisfazem as propriedades de segurança e vivacidade.

Hip 1 (Hipótese de Uso Correto) *Um nó pode chamar a função SafeSendToken à condição que ele possua o $REAL$ token. Depois de chamá-la ele não detém mais o token.*

Lema 1 *A $t = 0$, todas as propriedades acima descritas são satisfeitas.*

Prova. A função *Initialisation* é chamada a $t = 0$. $PSafetyCond(0)$: S_0 é o único detentor do token e não há nenhuma mensagem $\langle$*TOKEN*$\rangle$ pendente; $PHolderCount(0)$: o valor da variável *count* de todos os nós é igual a 0; $PHolderMonitored(0)$ e $PNestedMonitoring(0)$: os nós com um conjunto de detecção não vazio, $S_0 \ldots S_k$, são sucessivos no anel e monitoram nós entre S_0, o detentor do token, e ele próprio.

Lema 2 $\forall t \in \mathcal{C}, PSafetyCond(t) \wedge PHolderCount(t) \implies PHolderCount(t+1)$.

Prova.

- Se $t+1 \notin \mathcal{C}_d$: $PSafetyCond(t)$ assegura que existe no máximo um nó S_i com $Token(S_i, t) = REAL$ enquanto $PHolderCount(t)$, que S_i possui o maior valor de $count$. Além disso, Hyp.1 garante que S_i é o único que pode executar $SaveSendToken$ e incrementar *count* (line 17) a $t+1$. Consequentemente, quando S_i enviar a mensagem $\langle$*TOKEN*$\rangle$ a $t+1$, esta contém o maior valor possível de *count* e o novo detentor do token atribuirá este valor à sua própria variável *count* quando da recepção desta mensagem (linha 24).
- Se $t+1 \in \mathcal{C}_d$: Quando o nó S_d chama a função $UseBackup$, $\#(\mathcal{D}(S_d, t+1)) - 1$ é adicionado à sua variável *count* (linha 39). Como S_d monitorava $Holder(t)$ a t:

$$Count(Holder(t), t) - Count(S_d, t+1) < \#(\mathcal{D}(S_i, t)) - 1 \\ \implies Count(Holder(t), t) < Count(S_d, t+1).$$

  Logo, como $PHolderCount(t)$ assegura que $Holder(t)$ possui o maior valor de *count* a t, S_d detém o maior valor de *count* a $t+1$.

Lema 3 $\forall t \in \mathcal{C}, PSafetyCond(t) \wedge PHolderMonitored(t) \implies PNestedMonitoring(t)$.

Prova. Seja S_i e S_j dois nós cujos respectivos conjuntos de detectores são não vazio. Como por hipótese $PHolderMonitored(t)$ é verdadeiro e estes conjuntos de detectores são compostos de nós sucessivos (linha 25), então $\{Holder(t), .., S_i\} \in \mathcal{D}(S_i, t)$ e $\{Holder(t), .., S_j\} \in \mathcal{D}(S_j, t)$. Consequentemente, se $S_i \prec_t S_j$ (resp. $S_j \prec_t S_i$) em $\mathcal{T}(t)$, então $\{Holder(t), .., S_i\} \in \mathcal{D}(S_j, t)$ (resp. $\{Holder(t), .., S_j\} \in \mathcal{D}(S_i, t)$).

Lema 4 $\forall t \in \mathcal{C}, PSafetyCond(t) \wedge PHolderCount(t) \wedge PHolderMonitored(t) \wedge PNestedMonitoring(t) \implies PHolderMonitored(t+1)$.

Prova.

- Se $t+1 \notin \mathcal{C}_d$: Prova por contradição. Suponhamos que $PSafetyCond(t)$ e $PHolderCount(t)$ são verdadeiros, mas não $PHolderMonitored(t+1)$, ou seja, existe um nó S_j com $\mathcal{D}$ não vazio que não monitora $Holder(t+1)$. $PSafetyCond(t)$ e Hyp.1 asseguram que apenas S_i, o detentor do token a t, pode chamar a função $SafeSendToken$ a $t+1$ para enviar uma nova mensagem $\langle$*TOKEN*$\rangle$ a seus $k+1$ sucessores. Além disso, $PHolderMonitored(t)$ garante que S_j monitora o detentor do token a t. Assim, por construção (linha 25), se $S_i \in \mathcal{D}(S_j, t)$, S_j monitora todos os nós entre S_i e S_j no anel. Chegamos então a uma contradição pois o $Holder(t+1) = S_{i+1}$ é monitorado por S_j.
- Se $t+1 \in \mathcal{C}_d$: S_d não monitora nenhum nó correto pois todos os nós que pertencem a $\mathcal{D}(S_d, t)$ são falhos. Além disso, $PNestedMonitoring(t+1)$ garante que dois nós não falhos com $\mathcal{D} \neq \emptyset$, ao menos um deles monitora o outro. Consequentemente, S_d pertence a todo conjunto de detecção não vazio. Como $Holder(t+1) = S_d$, $PHolderMonitored(t+1)$ é verdadeiro.

Lema 5 $\forall t \in \mathcal{C}, PSafetyCond(t) \wedge PHolderCount(t) \wedge PNestedMonitoring(t) \wedge PHolderMonitored(t) \implies PSafetyCond(t+1)$.

Prova.

1. Se $t+1 \notin \mathcal{C}_d$:
   $PSafetyCond$ é satisfeita em t. Nós distinguimos então os dois casos seguintes:
   (a) existe um nó, S_i com um ***REAL*** token ($Token(S_i, t) = REAL$) e não há nenhuma mensagem $\langle$***TOKEN***$\rangle$ pendente. Hyp. 1 assegura que S_i é o único nó que pode chamar a função ***SafeSendToken*** e, consequentemente, enviar uma mensagem $\langle$***TOKEN***$\rangle$ a $t+1$. Como ***SafeSendToken*** garante que $Token(S_i, t_i + 1) = NONE$ (linha 19), $PSafetyCond$ é verdadeira a $t+1$.
   (b) não existe um nó com um ***REAL*** token mas há uma mensagem $\langle Pend_{REAL}\rangle$ pendente, endereçada a S_i, no sistema. Hyp. 1 garante que nenhum site pode mandar uma nova mensagem $\langle$***TOKEN***$\rangle$ a $t+1$, ou seja, a mensagem em questão é a única $\langle Pend_{REAL}\rangle$ do sistema. Consequentemente, $PSafetyCond$ é verdadeira a $t+1$: se S_i receber $\langle Pend_{REAL}\rangle$ a $t+1$, ele será o novo detentor do token; senão ele é o único receptor desta mensagem.
2. Se $t+1 \in \mathcal{C}_d$:
   Como $PHolderMonitored$ é verdadeira a t, $Holder(t)$ pertence a $\mathcal{D}(S_d, t)$. Distinguimos então os seguintes dois casos:
   - $Holder(t)$ é um nó de $\mathcal{D}(S_d, t)$ diferente de S_d. Como S_d regera um novo token apenas quando todos os nós que ele monitora, com exceção dele mesmo, falharem (linhas 29 e 36), não há nenhum outro nó correto com um $REAL$ token a $t+1$, ou seja, Holder(t) se encontra falho a $t+1$.
   - S_d é o $Holder(t)$. Como S_d não detém o $REAL$ token, existe uma $\langle Pend_{REAL}\rangle$ que lhe é endereçada. Neste caso, $PHolderCount(t+1)$ garante que o valor de *count* contido nesta mensagem não pode ser maior que o valor corrente *count* de S_d. O teste da linha 23 irá então ignorar esta mensagem.

   Em ambos os casos $PSafetyCond(t+1)$ é verdadeira.

Teorema 1 *Para o máximo de k falhas consecutivas, nosso algoritmo assegura a propriedade de segurança.*

Prova. A demonstração é consequência direta dos lemas anteriores

Teorema 2 *Para o máximo de k falhas consecutivas, nosso algoritmo assegura a propriedade de vivacidade.*

Prova. A fim de demonstrar a propriedade de vivacidade, basta provar que (1) se o token nunca se perder, todo site que detém o ***REAL*** token o enviará ao seu sucessor e (2) se o token é perdido devido à falha do detentor do ***REAL*** token, este será regerado pour um dos k nós sucessores do detentor que falhou.

A prova de (1) é trivial pois quando S_i envia $k+1$ cópias da mensagem $\langle$***TOKEN***$\rangle$ (linha 18) a t, ele informa nesta mensagem que o próximo detentor do token é o nó S_{i+1} ($Holder(t+1) = S_{i+1}$). Como os canais são confiáveis e S_i possui o maior valor de *count* a t ($PSafetyCond(t)$ e $PHolderCount(t)$), S_{i+1} terá o $REAL$ a $t+1$.

Para provar (2), consideremos que S_i é o último nó a possuir o token que enviou as $k+1$ cópias da mensagem $\langle$***TOKEN***$\rangle$ a t e que f é o número de sucessores falhos de

Si a $t+1$. Por hipótese, entre estes $k+1$ nós, há pelo menos um correto. Como $f \leq k$, o nó S_i enviou uma cópia da mensagem a $S_{i+f+1} \in \{S_{i+1}, .., S_{i+k+1}\}$. Além disso, como $PSafetyCond(t)$ e $PHolderCount(t)$ são verdadeiras pelas mesmas razões descritas em (1), S_{i+f+1} receberá a mensagem $\langle$*TOKEN*$\rangle$ enviada por S_i. A linha 25 atribuirá então $\{S_{i+1}, .., S_{i+f+1}\}$ a $\mathcal{D}$ de S_{i+f+1}. As falhas dos nós $\{S_{i+1}, .., S_{i+f}\}$ serão a termo detectadas, o que resultará na chamada da função *UseBackup* (linhas 30 e 37) que criará um novo *REAL* token.

3.4. Exemplos do Uso de Nosso Algoritmo

Discutimos nesta sub-seção sobre como alguns algoritmos existentes na literatura baseados na circulação do token e sua unicidade podem chamar as funções oferecidas por nosso algoritmo para que continuem corretos em presença de falhas dos nós. Consideramos que o sistema sobre o qual esses algoritmos executam são síncronos.

Nosso algoritmo se adapta naturalmente ao algoritmo proposto por Le Lann [Lann 1977] em que o acesso exclusivo ao recurso compartilhado é condicionado à possessão de um token, único, que circula entre todos os nós do sistema organizados em anel. Quando um processo recebe o token, se ele precisar acessar um recurso compartilhado, ele o faz. Senão, simplesmente repassa o token ao seu sucessor. Ao terminar o acesso ao recurso compartilhado, ele também envia o token ao seu sucessor. Para garantir a correta execução do algoritmo em presença de falhas, cada processo precisa simplesmente chamar as funções $SafeSendToken$ e $SafeReceiveToken$ oferecidas pelos nosso API de comunicação para respectivamente enviar e receber o token.

Alguns algoritmos de detecção do término de uma aplicação distribuída [Dijkstra et al. 1986] [Misra 1983] consideram os nós organizados logicamente em anel. O princípio destes algoritmos é verificar que todos os N nós se encontram passivos após uma volta completa do token no anel. No algoritmo de [Misra 1983], os nós possuem a cor branca (inicial) ou preta e um token circula entre os nós. A cor preta indica que o nó ficou ativo após a passagem do token. Este contém uma variável, *passif_count*, que contabiliza o número de nós que o token encontrou passivo (cor branca). Um nó inicia o algoritmo de detecção. Ao receber o token, se o nó receptor é de cor branca, ele reinicializa *passif_count* a 1, senão ele o incrementa. O término é detectado quando todos os nós são de cor branca após uma volta total do token ($passif\_count = N$). Este algoritmo pode utilizar as funções $SafeSendToken$ e $SafeReceiveToken$ para assegurar a circulação do token. Um nó falho pode ser visto como passivo. Assim, se o novo detentor do $REAL$ token não o recebeu de seu predecessor (falha de um ou mais nós), o valor do contador do token precisa ser atualizado antes de ser entregue ao algoritmo de término. A função $UpdateToken()$ teria então o seguinte código:

```
43 UpdateToken (⟨TOKEN⟩)
44 ⌊ ⟨TOKEN⟩.passif_count ← ⟨TOKEN⟩.passif_count + ( #(D) - 1 )
```

Note que consideramos que (1) o nó iniciador do algoritmo só pode falhar após ter enviado as $k+1$ cópias da mensagem $\langle$*TOKEN*$\rangle$ e que (2) no caso de falhas, o sistema de monitoramento só indica a ocorrência de uma ou mais falhas (função $ReceiveSuspected()$) após esperar um certo intervalo de tempo que assegure que não existe mensagens pendentes para o novo detentor do token quando este o entregar ao

algoritmo de término.

Vários autores [Chang and Roberts 1979], [Franklin 1982], [Peterson 1982], etc. propuseram algoritmos para o problema da eleição de um líder para nós interligados em uma estrutura lógica de anel. Os nós candidatos a se tornarem líder enviam uma mensagem de candidatura (token) ao seu sucessor que circula no anel segundo as regras de comparação e transmissão do algoritmo em questão. Por exemplo, no algoritmo de Chang e Roberts, a mensagem é retransmitida enquanto um melhor candidato não é encontrado. Um nó é eleito líder quando receber a mensagem que contém a sua própria candidatura.

Para assegurar a correta execução do algoritmo de Chang e Roberts na presença de falhas, a circulação e a unicidade das mensagens de candidaturas precisam ser garantidas. Entretanto, uma mensagem de candidatura pode ser vista, analogamente ao token, como um objeto único que circula no anel. Em outras palavras, podemos aplicar nosso algoritmo para garantir a tolerância a falha dos pedidos de candidatura: a mensagem $< TOKEN >$ representa então a candidatura de um nó cuja identificação S_l está contida na mensagem. Como para um anel com N nós, o número máximo de pedidos pendentes é N, nosso algoritmo precisa controlar a circulação e unicidade de no máximo N objetos. Observe que consideramos a mesma hipótese (2) de que não há mensagens pendentes para o novo detentor do objeto no caso de falhas de seus predecessores.

O nosso algoritmo assegura que em presença de no máximo $k + 1$ falhas consecutivas, o algoritmo de eleição termina (se e somente se ao menos um dos candidatos emitiu $k + 1$ cópias da mensagem de candidatura). Vale ressaltar que não podemos assegurar que o líder seja um processo correto. Entretanto, poderíamos oferecer uma função $UpdateToken$ para tanto. O nó S_j se torna líder ao receber sua própria mensagem de candidatura. Esta foi enviada com a utilização da função $SafeSendToken$ e, consequentemente, também recebida pelos k sucessores de S_j no anel, que monitoram então S_j. No caso de falha deste, a função $UpdateToken()$ é executada por S_i, o sucessor correto de S_j. Este sabe que se trata da falha de um líder pois o havia registrado como tal (*currentLeader*). Assim, para assegura a eleição de um nó correto basta alterar a candidatura de S_j pela de S_i. Ao receber a sua propria candidatura S_i se tornará então o novo líder. O código da função $UpdateToken()$ seria:

```
45 UpdateToken (⟨TOKEN⟩)
46     if currentLeader = S_l then
47         set S_i as S_l in ⟨TOKEN⟩
```

4. Trabalhos Relacionados

Vários algoritmos de exclusão mútua tolerante a falhas [Nishio et al. 1990] [Manivannan and Singhal 1994] [Chang et al. 1990], etc. existem na literatura. Porém, estes geralmente adotam soluções que não são escaláveis como por exemplo uma eleição global ou necessitam que o nó que detectou a perda do token receba um confirmação (*acknowledge*) de todos os outros nó antes de recriar o token como em [Nishio et al. 1990] [Manivannan and Singhal 1994].

Misra [Misra 1983] propõe em seu artigo sobre detecção do término de uma aplicação distribuída para topologias em anel adaptar o algoritmo para que a perda do

token possa ser detectada e un novo token então regerado. O autor argumenta que a perda do token é similar à detecção de término da aplicação se esta considerar apenas as mensagens ligadas ao token. Como nossa solução, o autor usa o conceito de ter mais do que um token: há dois token simétricos no sistema mas um deles é visto como o token backup. Comparando o número de sequência que um token possui com o armazenado pelos nós, um token pode detectar a perda do outro. Entretanto, a detecção é possível somente se aquele também não se perder na mesma volta que este (*round*).

Em [Mueller 2001], Mueller apresenta um mecanismo que oferece tolerância a falhas para protocolos de sincronização baseados em token. Um anel lógico é utilizado para detectar a falha de um nó e, se necessário, eleger um novo detentor do token. Porém, contrariamente à nossa solução, a detecção e tratamento das falhas não é transparente à aplicação. Esta precisa ser modificada para incluir um sistema de monitoramento de nós e este então, ao suspeitar uma falha, chama o mecanismo tolerante a falhas proposto pelo autor. Além disso, este mecanismo organiza os nós em um anel lógico que permite detectar a falha de apenas um único nó e cuja manutenção é extremamente cara, o que limita a escalabilidade da solução. Um terceiro ponto é que ao criar um novo token, o estado do token não é preservado ou restaurado, como na nossa solução.

Inúmeros algoritmos de difusão de mensagens utilizam um mecanismo baseado em token com nós organizados em anel para ordenar mensagens emitidas por difusão (*total order broadcast*). O token circula entre os nós ou um subconjunto de nós. Alguns algoritmos como [Chang and Maxemchuck 1984] e [Amir et al. 1995] toleram falhas de nós mas envolvem mecanismos caros e globais como uma fase de reconstrução em que a difusão não é permitida ou um protocolo de gestão de filiação de grupo para reconstruir o anel e eleger um novo detentor to token.

O trabalho de Ekwall et al. [Ekwall et al. 2004] é proximo ao nosso no sentido em que os nós do sistema são organizados em um anel lógico e o token é enviado a $f + 1$ nós sucessores, sendo f o número máximo de falhas. Porém, o objetivo dos autores é diferente do nosso. Eles consideram um sistema assíncrono sobre o qual querem construir um algoritmo de consenso baseado em token. Um nó espera receber o $token$ de seu predecessor. Porém, se aquele suspeitar que este falhou ele espera o token de qualquer um de seus f predecessores. Contrariamente à nossa solução, a detecção não é perfeita e a unicidade do token não é garantida. Além disso, a proposta de nosso algoritmo é que ele possa ser utilizado por outros algoritmos baseados em circulação do token em anel que necessitem garantir a unicidade do token em caso de falhas. Esta portabilidade não é oferecida na solução dos autores que se concentram no problema do consensus et difusão atômica (*atomic broadcast*). Uma última observação é que nosso algoritmo tolera k falhas consecutivas e não apenas k, ou seja, o número de falhas pode ser maior que k.

5. Conclusão

Apresentamos neste artigo um algoritmo que evita a perda do token enviando cópias deste a $k + 1$ nós. Nossa solução tolera até k falhas consecutivas. As funções oferecidas podem ser facilmente utilizadas por algoritmos existentes organizados logicamente em anel que precisem assegurar a unicidade do token, mas que não tratam o problema da perda do token em caso de falhas. Tanto a detecção de falhas como a criação de um novo token são transparentes a este algoritmo e implementadas de forma eficaz e escalável: um nó monitora apenas seus k nós predecessores e a criação de um token é praticamente

instantânea. Além disso, o anel não precisa ser reconstruído e, se o token contiver dados da aplicação, estes podem ser mais facilmente recuperados.

Referências

Amir, Y., Moser, L. E., Melliar-Smith, P. M., Agarwal, D. A., and Ciarfella, P. (1995). The totem single-ring ordering and membership protocol. *ACM Trans. Comput. Syst.*, 13(4):311–342.

Chang, E. and Roberts, R. (1979). An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Commun. ACM*, 22(5):281–283.

Chang, I., Singhal, M., and Liu, M. T. (1990). A fault tolerant algorithm for distributed mutual exclusion. In *Proceedings of the IEEE 9th Symposium on Reliable Distributed Systems*, pages 146–154.

Chang, J.-M. and Maxemchuck, N. F. (1984). Reliable broadcast protocols. *ACM Trans. Comput. Syst.*, 2(3):351–273.

Défago, X., Schiper, A., and Urbán, P. (2004). Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421.

Dijkstra, E. W., Feijen, W. H. J., and van Gasteren, A. J. M. (1986). Derivation of a termination detection algorithm for distributed computations. In *Proc. of the NATO Advanced Study Institute on Control flow and data flow: concepts of distributed programming*, pages 507–512.

Ekwall, R., A.Schiper, and Urbán, P. (2004). Token-based atomic broadcast using unreliable failure detectors. In *SRDS*, pages 52–65.

Franklin, R. W. (1982). On an improved algorithm for decentralized extrema finding in circular configurations of processors. *Communications of the ACM*, 25(5):336–337.

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

Manivannan, D. and Singhal, M. (1994). An efficient fault-tolerant mutual exclusion algorithm for distributed systems. In *ISCA International Conference on Parallel and Distributed Computing Systems*, pages 525–530.

Misra, J. (1983). Detecting termination of distributed computations using markers. In *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 290–294.

Mueller, F. (2001). Fault tolerance for token-based synchronization protocols. *Workshop on Fault-Tolerant Parallel and Distributed Systems, IEEE*.

Nishio, S., Li, K. F., and Manning, E. G. (1990). A resilient mutual exclusion algorithm for computer networks. *IEEE Trans. on Parallel and Distributed Systems*, 1(3):344–355.

Peterson, G. L. (1982). An o(nlog n) unidirectional algorithm for the circular extrema problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(4):758–762.

Um Serviço Distribuído de Detecção de Falhas Baseado em Disseminação Epidêmica

Leandro P. de Sousa, Elias P. Duarte Jr.

Departamento de Informática – Universidade Federal Paraná (UFPR)
Caixa Postal 19018 – 81531-980 – Curitiba – PR

{leandrops,elias}@inf.ufpr.br

Abstract. *Failure detectors are abstractions that can be used to solve consensus in asynchronous systems. This work presents a failure detection service based on a gossip strategy. The service was implemented on the JXTA platform. A simulator was also implemented so the detector could be evaluated for a larger number of processes. Experimental results show CPU and memory usage, fault and recovery detection time, mistake rate and how the detector performs when used in a simple election algorithm. The results indicate that the service scales well as the number of processes grow.*

Resumo. *Detectores de falhas são abstrações que, dependendo das propriedades que oferecem, permitem a solução do consenso em sistemas distribuídos assíncronos. Este trabalho apresenta um serviço de detecção de falhas baseado em disseminação epidêmica. O serviço foi implementado para a plataforma JXTA. Para permitir a avaliação com um número maior de processos, foi também implementado um simulador. Resultados experimentais são apresentados para o uso de processador e memória, tempo de detecção, taxa de enganos do detector, além do seu uso na execução de eleição de líder. Os resultados experimentais e de simulação indicam que o serviço é escalável com o número de processos e mostram que a estratégia de disseminação epidêmica possui vantagens significativas em grupos com grande número de processos.*

1. Introdução

Grande parte dos problemas relacionados aos sistemas distribuídos requer algum tipo de coordenação entre os diversos componentes [Greve 2005, Turek and Shasha 1992], chamados aqui de *processos*. Tanto os processos quanto os canais de comunicação entre eles podem sofrer falhas. Desse modo, um requisito fundamental para a coordenação entre os processos é que estes conheçam os estados uns dos outros, para que estes possam tomar as providências necessárias em caso de falha. Em alguns tipos de sistemas distribuídos, esta pode ser uma tarefa difícil ou mesmo impossível. Este é o caso dos *sistemas assíncronos*. Neste tipo de sistema, tanto os processos quanto os canais de comunicação podem se comportar de maneira arbitrariamente lenta, tornando um processo falho indistinguível de um processo muito lento. Este problema é conhecido na literatura como a *impossibilidade FLP* [Fischer et al. 1985]. Este resultado torna impossível a resolução de uma classe de problemas distribuídos de extrema importância, os chamados *problemas de acordo*.

Como uma maneira de contornar a *impossibilidade FLP* e tornar possível a resolução do consenso, Chandra et al. criaram os *detectores de falhas não-confiáveis*

[Chandra and Toueg 1996]. É fato que, em sistemas completamente assíncronos, detectores de falhas que forneçam alguma garantia de completude e exatidão *não podem ser implementados*. Ainda assim, o estudo da implementação destes detectores é de grande importância. Algoritmos que fazem o uso de um detector são mais genéricos, pois não precisam se preocupar com as característas temporais do sistema.

Este trabalho apresenta a especificação e implementação de um serviço de detecção de falhas baseado em disseminação epidêmica. O protocolo de detecção de falhas é probabilístico, se utilizando de uma estratégia de envio de *gossips* [Gupta et al. 2002]. Para utilizar o detector, um processo precisa implementar o serviço e participar de um grupo de detecção. A qualquer momento, um processo pode consultar seu detector e obter uma lista de processos considerados falhos e corretos. O funcionamento do algoritmo pode ser alterado através de parâmetros do serviço. O serviço de detecção foi implementado para a plataforma JXTA [JXTA 2009]. Um simulador também foi implementado, utilizando a biblioteca SMPL [MacDougall 1997]. Resultados experimentais são apresentados tanto para a implementação JXTA quanto para a simulação. São avaliados o uso de recursos do sistema, probabilidade de enganos e velocidade do detector para diferentes configurações de parâmetros do serviço de detecção.

Os resultados da simulação indicam que o serviço proposto é escalável para o número de processos, tanto em relação à troca de mensagens quanto à qualidade da detecção de falhas. Estes resultados também mostram que a estratégia de disseminação epidêmica é robusta e apresenta vantagens significativas em grupos com grande número de processos. Porém, os experimentos realizados para a plataforma JXTA apontam para o uso elevado de processamento pela implementação do serviço, mesmo para um grupo pequeno de processos.

Este trabalho está organizado da seguinte maneira. Na seção 2, são apresentados alguns trabalhos relacionados na área de implementação de detectores de falha. Na seção 3, são descritos o serviço proposto juntamente com o algoritmo de deteção utilizado. Na seção 4 são apresentados resultados experimentais e de simulação. A seção 5 conclui o trabalho e apresenta algumas considerações sobre trabalhos futuros.

2. Trabalhos Relacionados

Detectores de falhas não podem ser implementados em sistemas assíncronos, devido à impossibilidade FLP. Ainda sim, o estudo de detectores de falhas do ponto de vista prático é de extrema importância na implementação de soluções para os problemas de sistemas distribuídos. Em [Chandra and Toueg 1996], os autores mostram que detectores de falhas não-confiáveis podem ser implementados em sistemas de sincronia parcial. No modelo de sistema adotado pelos autores, existem limites superiores tanto para transmissão de mensagens quanto para o tempo de execução dos processos, mas estes são desconhecidos e só valem após um tempo de estabilização, chamado GST. Desse modo, uma implementação de um detector que faz o uso de *timeouts* pode, após um tempo, passar a detectar a ocorrência de falhas dos processos.

Em sistemas reais, o modelo de sincronia parcial teórico não é respeitado. Porém, estes sistemas normalmente alternam entre períodos de estabilidade e instabilidade. Durante um período de instabilidade, o sistema é completamente assíncrono. Já durante um período de estabilidade, pode-se dizer que o sistema respeita algumas propriedades tem-

porais, assim detectores de falhas são possíveis. Se este período for longo o suficiente, problemas como o do consenso podem ser resolvidos nestes sistemas [Raynal 2005].

Para o uso prático de detectores de falhas, aplicações podem ter necessidades além das propriedades eventuais dos detectores, propostas em [Chandra and Toueg 1996]. Aplicações podem ter restrições temporais, e um detector que possui um atraso muito grande na detecção de falhas pode não ser suficiente. Por este motivo, [Chen et al. 2002] propõe métricas para a *qualidade de serviço (quality of service)*, ou simplesmente QoS, de detectores de falhas. De maneira geral, as métricas de QoS para um detector de falhas buscam descrever a *velocidade (speed)* e a *exatidão (accuracy)* da detecção. Em outras palavras, as métricas definem quão rápido o detector detecta uma falha e quão bem este evita enganos. Ainda em [Chen et al. 2002], os autores propõem um detector de falhas, chamado NFD-E, que pode ser configurado de acordo com os parâmetros de QoS necessários para a aplicação em questão. Este detector visa o sistema probabilístico proposto pelos autores.

Em [Das et al. 2002], é descrito um protocolo de gestão da composição de grupos, chamado SWIM. O protocolo SWIM é dividido em duas partes: um detector de falhas e um protocolo de disseminação da composição do grupo. Este detector de falhas foi proposto inicialmente em [Gupta et al. 2001]. O detector utiliza uma estratégia de *ping* randomizado, onde cada processo periodicamente testa outro processo, selecionado aleatoriamente. Informações sobre a composição do grupo e falhas de processos são transmitidas nas próprias mensagens do detector de falhas, através de um mecanismo de *piggybacking*.

Em [van Renesse et al. 1998], é proposto um algoritmo de detecção de falhas que se utiliza de uma estratégia de disseminação epidêmica (*gossip*). De acordo com esta estratégia, processos enviam periodicamente mensagens de *gossip* para um grupo de outros processos, estes escolhidos de maneira aleatória. A detecção falhas é feita por um mecanismo de *heartbeat*. Cada mensagem de *gossip* é composta pelo valor de *heartbeat* do processo emissor juntamente com últimos valores de *heartbeat* que este tenha recebido de outros processos. Processos que não recebem informação nova sobre um outro determinado processo, após um determinado intervalo de tempo, passam a considerar este último como falho. Os autores sugerem ainda uma melhora para o algoritmo, para o caso do mesmo ser utilizado em um ambiente como o da Internet. Mais especificamente para o caso dos processos estarem espalhados em diferentes domínios e sub-redes. O serviço de deteção proposto neste trabalho se baseia neste algoritmo de deteção.

3. O Serviço de Detecção Proposto

Este trabalho propõe um serviço de deteção de falhas baseado em disseminação epidêmica. O serviço foi implementado para a plataforma JXTA, sendo chamado JXTA-FD. O algoritmo de detecção implementado pelo serviço JXTA-FD é baseado no algoritmo proposto em [van Renesse et al. 1998]. O monitoramento entre os processos do sistema utiliza uma estratégia de *heartbeats*, que são propagados através de disseminação epidêmica (*gossip*). O algoritmo completo do serviço pode ser visto na Figura 1.

O sistema considerado para a execução do algoritmo é representável por um grafo completo, isto é, cada processo pode se comunicar com qualquer outro processo. Esta consideração foi feita com base em funcionalidades providas pela plataforma JXTA. O

Every JXTA-FD instance executes the following:

|| *Initialization:*
$table \leftarrow$ **new** HBTable
$heartbeat \leftarrow 0$
$timeOfLastBcast \leftarrow 0$
start tasks *ReceiverTask*, *GossipTask*, *BroadcastTask* and *CleanupTask*

|| *ReceiverTask:* **whenever a gossip message** m **arrives**
for all <*ID*, *hbvalue*> $\in m$ **do**
  $table$.*update*(*ID*, *hbvalue*)
end for
if m is a broadcast **then**
  $timeOfLastBcast \leftarrow$ current time
end if

|| *GossipTask:* **repeat every** $GOSSIP\_INTERVAL$ **units of time**
if $table$ is not empty **then**
  $numberOfTargets \leftarrow min(FANOUT, table.size())$
  $targets \leftarrow$ choose $numberOfTargets$ random elements from $table.get\_ids()$
  for all $t \in targets$ **do**
    send gossip message to t
  end for
  $heartbeat \leftarrow heartbeat + 1$
end if

|| *BroadcastTask:* **repeat every** $BCAST\_TASK\_INTERVAL$ **units of time**
if $shouldBcast()$ **then**
  send gossip message by broadcast
  $timeOfLastBcast \leftarrow$ current time {not necessary if the process receives its own broadcasts}
end if

|| *CleanupTask:* **repeat every** $CLEANUP\_INTERVAL$ **units of time**
for all $id \in table.get\_ids()$ **do**
  $timeFromLastUpdate \leftarrow$ current time - $table.get\_tstamp$(*ID*)
  if $timeFromLastUpdate \geq REMOVE\_TIME$ **then**
    remove id from $table$
  end if
end for

Figura 1. Algoritmo de detecção de falhas do serviço JXTA-FD.

sistema é assíncrono; mais especificamente, o sistema tem propriedades probabilísticas tanto para o atraso dos canais de comunicação quanto para a falha de processos. Falhas de processos são falhas por parada. Processos que falham podem retornar ao sistema com um novo identificador. Cada processo executa uma instância do algoritmo de detecção.

O funcionamento do algoritmo é divido em quatro rotinas que executam em paralelo: *ReceiverTask*, *GossipTask*, *BroadcastTask* e *CleanupTask*, além de um procedimento de inicialização. As seções seguintes detalham as estruturas de dados, rotinas do algoritmo e funcionamento do detector.

3.1. Estruturas de Dados

A principal estrutura de dados utilizada pelo algoritmo é chamada $HBTable$. Uma $HBTable$ tem por finalidade armazenar valores de *heartbeat* de outros processos juntamente com o tempo da última atualização de cada um. Cada *heartbeat* é representado por um valor inteiro positivo. Uma $HBTable$ é implementada como uma tabela *hash* que utiliza como chave um identificador de processo, aqui chamado de ID, e como valor uma tupla composta por dois valores inteiros, um representando o valor do $heartbeat$ e o outro o $timestamp$ da última atualização. A notação <*hbvalue*, *tstamp*> é utilizada para representar esta tupla.

Uma $HBTable$ fornece cinco operações básicas: *update*(*ID*, *hbvalue*), *get_hbvalue*(*ID*), *get_tstamp*(*ID*), *size*() e *get_ids*(). A operação *update*(*ID*, *hbvalue*), quando invocada, primeiro verifica se o valor $hbvalue$ passado é maior do que o armazenado para aquele ID. Se sim, o novo valor é armazenado e o $timestamp$ correspondente é alterado para o tempo atual. Caso a $HBTable$ não possua um valor para o ID passado, uma tupla com o $hbvalue$ passado e o tempo atual é inserida na estrutura. As operações *get_hbvalue*(*ID*) e *get_tstamp*(*ID*) retornam os valores de $hbvalue$ e $tstamp$, respectivamente, para um dado ID. Por fim, *size*() retorna o número de entradas na tabela e *get_ids*() retorna o conjunto dos IDs contidos na mesma.

Cada instância do algoritmo faz uso de uma $HBTable$ e de dois inteiros, $localHB$ representando o valor atual de $heartbeat$ do processo local e $timeOfLastBcast$ representando o tempo no qual foi recebida a última mensagem de difusão. $timeOfLastBcast$ é utilizado na decisão de quando um processo deve fazer uma nova difusão ou não.

3.2. Inicialização

Ao início da execução do algoritmo de detecção, um procedimento de inicialização é executado. Este procedimento tem como objetivo inicializar as estruturas de dados necessárias e disparar as outras rotinas do algoritmo de detecção. Primeiramente, uma $HBTable$ é criada, o valor de *heartbeat* local e o valor $timeOfLastBcast$ são inicializados para 0. Em seguida, as demais rotinas do algoritmo são iniciadas e passam a executar paralelamente.

3.3. ReceiverTask

A rotina *ReceiverTask* é executada toda vez que uma mensagem de *gossip* é recebida, inclusive no caso das mensagens ocasionais enviadas por difusão. Cada mensagem de *gossip* é composta de um conjunto de tuplas <*ID*, *hbvalue*>, sendo que cada uma representa o $heartbeat$ de um dado processo. Quando uma mensagem de *gossip* chega, a operação *update*(*ID*, *hbvalue*) da $HBTable$ é chamada para cada uma das tuplas, atualizando as informações de $heartbeat$ contidas na tabela. Quando a mensagem recebida é de difusão, o processo também incrementa o valor $timeOfLastBcast$.

3.4. GossipTask

A rotina *GossipTask* é executada periodicamente, a cada $GOSSIP\_INTERVAL$ intervalos de tempo. Ela é responsável pelo envio das mensagens de *gossip* aos outros processos. A cada execução, a rotina primeiramente verifica se $HBTable$ está vazia. Se sim, não há nada a ser feito, pois nenhum outro processo é conhecido. No caso de existir alguma entrada na tabela, $FANOUT$ processos são escolhidos aleatoriamente do conjunto de processos conhecidos (ou menos, no caso de o número de entradas na $HBTable$ ser insuficiente). Uma mensagem de *gossip* é enviada para cada um dos processos selecionados. A mensagem de *gossip* enviada é construída com as informações contidas na $HBTable$. Para cada entrada na tabela, uma tupla <*ID*, *hbvalue*> é adicionada à mensagem. Também é adicionada uma tupla com o ID e $heartbeat$ do processo local. Após o envio da mensagem, o processo incrementa seu valor local de $heartbeat$.

3.5. BroadcastTask

Para que os processos possam se encontrar inicialmente, e para que a saída do detector se estabilize mais rapidamente em caso de um número alto de falhas simultâneas, a rotina *BroadcastTask* é executada periodicamente. A cada execução, existe a chance de uma mensagem de *gossip* do algoritmo ser enviada para todos os outros processos, através de um mecanismo de difusão. A probabilidade da difusão ser efetuada é calculada com base em parâmetros do algoritmo e no tempo de chegada da última mensagem de difusão recebida. Esta probabilidade visa evitar difusões simultâneas e muito frequentes.

Para a implementação do algoritmo, o JXTA-FD utiliza a mesma função de probabilidade proposta em [van Renesse et al. 1998], $p(t) = (t/BCAST\_MAX\_PERIOD)^{BCAST\_FACTOR}$, onde t é igual ao número de unidades de tempo decorridas da última difusão recebida e $BCAST\_MAX\_PERIOD$ e $BCAST\_FACTOR$ são parâmetros do algoritmo. A cada execução da rotina *BroadcastTask* uma mensagem de *gossip* é enviada por difusão com probabilidade $p(t)$. Desse modo, o tempo médio entre o envio de difusões depende diretamente da frequência de execução da rotina (controlada pelo parâmetro $BCAST\_TASK\_INTERVAL$), do número de processos no sistema e dos parâmetros do algoritmo. $BCAST\_MAX\_PERIOD$ é o intervalo máximo entre cada difusão, e quando t se aproxima deste valor, a probabilidade $p(t)$ tende a 1. $BCAST\_FACTOR$ deve ser um valor real positivo, e determina o quão antes de $BCAST\_MAX\_PERIOD$ as difusões tendem a ser enviadas. Quanto maior o valor de $BCAST\_FACTOR$, mais próxima do valor $BCAST\_MAX\_PERIOD$ fica a duração do intervalo esperado entre as difusões.

Como exemplo, para os valores $BCAST\_TASK\_INTERVAL$ de 1 unidade de tempo, $BCAST\_MAX\_PERIOD$ de 20 unidades de tempo e um conjunto de 1000 processos, para se obter um intervalo de aproximadamente 10 unidades de tempo entre uma difusão e a próxima, $BCAST\_FACTOR$ deve ser aproximadamente 10.43 [van Renesse et al. 1998].

3.6. CleanupTask

A rotina *CleanupTask* é responsável por remover entradas antigas da $HBTable$ do processo local. A cada $CLEANUP\_INTERVAL$ unidades de tempo, a tabela é percorrida e entradas que não foram atualizadas a mais de $REMOVE\_TIME$ unidades de

tempo são excluídas. Este mecanismo é importante pois processos considerados suspeitos também são utilizados pelo mecanismo de *gossip* como possíveis alvos, e uma tabela contendo um número muito alto de entradas inválidas (processos falhos) pode causar um impacto negativo na exatidão do detector. O valor $REMOVE\_TIME$ é um parâmetro de configuração do algoritmo.

3.7. Saída do Detector

A qualquer momento durante a execução do algoritmo, um processo pode consultar seu detector local pelo conjunto de processos suspeitos ou corretos. Para determinar estes processos, a tabela $HBTable$ é percorrida e, para cada entrada, o tempo decorrido desde a sua última atualização é calculado. Se este tempo for maior ou igual a $SUSPECT\_TIME$, o processo é considerado suspeito. Caso contrário, ele é considerado correto.

4. Implementação e Resultados Experimentais

O serviço JXTA-FD foi implementado como um módulo (*Module*) para a plataforma JXTA, versão 2.5, utilizando a linguagem Java. O monitoramento entre *peers* é feito dentro do contexto de um *Peer Group*, e um módulo do JXTA-FD deve ser carregado e inicializado para cada grupo monitorado. Somente *peers* que estejam executando o módulo participam do algoritmo. A qualquer momento, um *peer* pode consultar seu módulo de detecção pela lista de processos corretos ou suspeitos.

O comportamento do serviço pode ser controlado através de alguns parâmetros de configuração. Os parâmetros mais importantes são três: $GOSSIP\_INTERVAL$, $FANOUT$ e $SUSPECT\_TIME$. O primeiro controla o intervalo entre o envio de mensagens de *gossip* pela rotina *GossipTask*. O segundo controla quantas mensagens são enviadas em cada execução da mesma. Por último, $SUSPECT\_TIME$ determina quantas unidades de tempo sem atualização de $heartbeat$ são necessárias para se considerar um *peer* como suspeito. Os parâmetros do serviço para um dado grupo devem ser decididos a priori, antes da inicialização do mesmo.

4.1. Avaliação e Resultados Experimentais

Para a avaliação do serviço de detecção de falhas proposto, experimentos foram realizados para a implementação na plataforma JXTA e para um simulador, implementado com o uso da biblioteca de eventos discretos SMPL [MacDougall 1997].

Nos experimentos realizados, foram avaliadas duas estratégias diferentes para a configuração do detector. Na primeira, a cada rodada do algoritmo é enviada apenas uma mensagem de *gossip*. Para aumentar a exatidão do detector, o intervalo entre o envio das mensagens de *gossip* (parâmetro $GOSSIP\_INTERVAL$) é reduzido. Na segunda estratégia, o intervalo entre o envio de mensagens de *gossip* é mantido fixo enquanto o *fanout* do algoritmo (parâmetro $FANOUT$), ou seja, o número de mensagens de *gossip* enviadas a cada rodada, é incrementado. As comparações são feitas de modo que a banda utilizada, isto é, a quantidade de tuplas do tipo $<$*ID*, *hbvalue*$>$ enviada pelos peers em um dado intervalo de tempo, seja a mesma para os dois casos. Estas estratégias são representadas nos gráficos por *Gossip* e *Fanout*, respectivamente.

Para a simulação do atraso e perda de mensagens, foi implementado como um parâmetro da execução dos experimentos um valor que controla a porcentagem de mensagens perdidas pelos *peers*. Cada mensagem tem uma chance de ser descartada. Esse

mecanismo foi adotado para simplificar a implementação e a avaliação dos resultados, visto que mensagens suficientemente atrasadas causam o mesmo impacto de mensagens perdidas no funcionamento do detector.

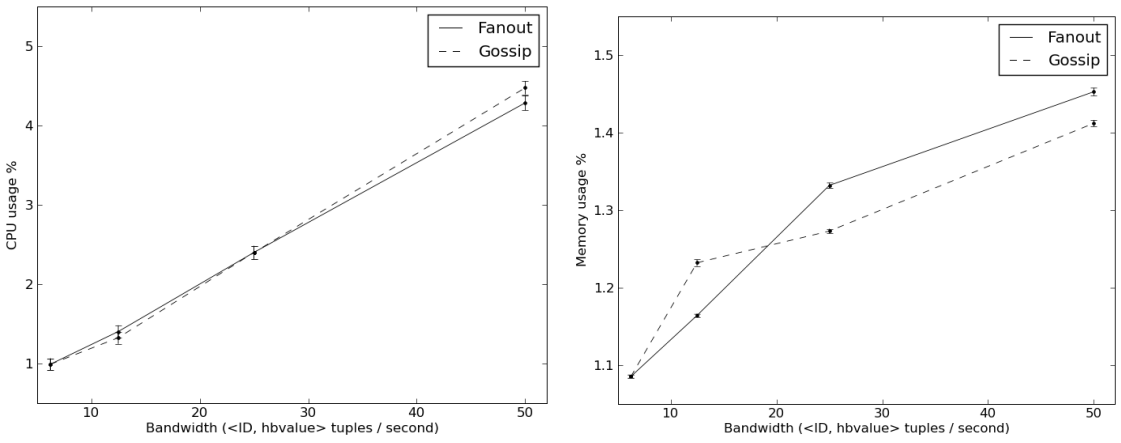
4.2. Resultados da Implementação JXTA

Os experimentos foram realizados para um grupo de *peers* executando em um único *host* e o mecanismo de descarte de mensagens é utilizado para representar o atraso e a perda de mensagens. Os gráficos destes experimentos apresentam os resultados obtidos com um intervalo de confiança de 95%.

Uso de CPU e Memória

O objetivo destes experimentos é avaliar o uso de CPU e memória pelo serviço de detecção e plataforma JXTA. Os experimentos foram realizados em uma máquina Intel Core2 Quad Q9400, 2.66GHz, com 4GB de memória RAM. Os experimentos foram realizados com 10 *peers* executando o serviço de detecção por 15 minutos. A cada 1 segundo, são obtidos os dados relacionados ao uso de CPU e memória. Todos os *peers* executam o detector com os mesmos parâmetros. O tempo para suspeitar de um *peer*, $SUSPECT\_TIME$ é de 5 segundos. O tempo para remoção de um *peer* suspeito, $REMOVE\_TIME$, é de 20 segundos. Cada *peer* realiza 1 consulta por segundo ao seu detector. Os parâmetros $BCAST\_MAX\_PERIOD$ e $BCAST\_FACTOR$ são 20 e 4.764 respectivamente.

A Figura 2(a) mostra o impacto dos parâmetros do detector no uso de CPU. O gráfico demonstra que o uso de CPU é diretamente proporcional ao uso de banda pelo serviço de detecção. Nos testes realizados, pouca diferença é observada entre as duas estratégias do detector. É possível que, dado um grupo maior de *peers*, a diferença entre as duas estratégias se torne mais expressiva. Do mesmo modo, pode-se observar na Figura 2(b) que o uso de memória também acompanha o uso de banda do detector.



(a) (b)

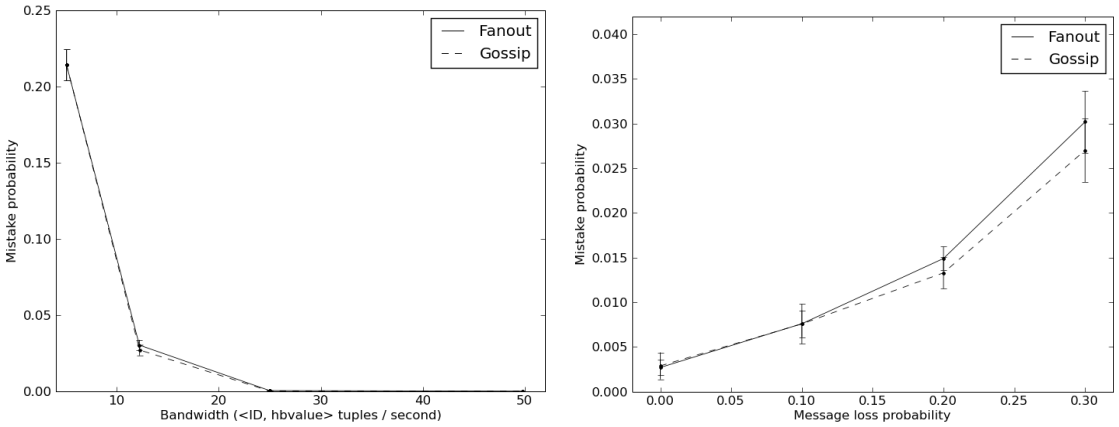
Figura 2. (a) Impacto dos parâmetros do detector no uso de CPU. (b) Impacto dos parâmetros do detector no uso de memória.

Estes resultados demonstram que a plataforma JXTA consome uma quantidade

de recursos considerável, tendo em vista que este é um serviço básico utilizado para a implementação de outros algoritmos. Vale ressaltar que a quantidade de *peers* utilizada é bastante pequena e estes participam de apenas um grupo de detecção.

Probabilidade de Enganos

Estes experimentos buscam verificar o impacto dos parâmetros do detector e do número de falhas no número de enganos do detector. Um engano ocorre quando um *peer* correto é suspeito pelo detector. Nestes experimentos, os *peers* nunca falham. Desse modo, qualquer suspeita do detector é um engano. O cenário dos experimentos é o mesmo dos experimentos de uso de CPU, os testes são executados para 10 *peers*, e os valores dos parâmetros fixos são os mesmos.



(a) (b)

Figura 3. (a) Variação no número de enganos de acordo com a banda utilizada. Experimento realizado com 30% das mensagens sendo descartadas. (b) Variação no número de enganos de acordo com a probabilidade de perda de mensagens. O valor de banda utilizada é fixado em 25.

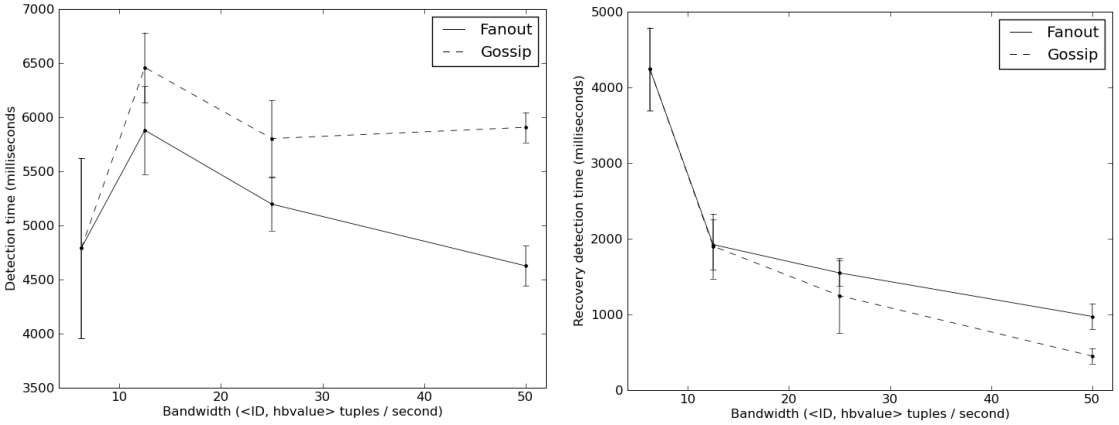
A Figura 3(a) mostra a probabilidade de uma consulta ao detector ser um engano, para uma quantidade de perda de mensagens igual a 30%. É possível observar a melhora na exatidão do detector com o aumento do *fanout* ou na frequência do envio de mensagens de *gossips*. A probabilidade de enganos para o valor de banda 12.5 é aproximadamente 0.02700 para a estratégia *Gossip* e 0.03019 para a estratégia *Fanout*. Para o valor de banda 25, os valores são 0.00015 e 0.00035, respectivamente. Para o valor de banda 50, não foram detectados enganos. Pode-se observar que, para um grupo tão pequeno de *peers*, a diferença entre as duas estratégias não é muito expressiva. Apesar disso, a estratégia de *Gossip* obteve um resultado melhor, com menos de 50% na quantidade de enganos para o valor de banda 25.

Na Figura 3(b) é mostrado o impacto da probabilidade de perda de mensagens no número de enganos cometidos pelo detector. Os resultados demonstram que a estratégia *Gossip* é mais resistente à perda de mensagens. Para os valores de 20% e 30% de mensagens perdidas, a quantidade de enganos é aproximadamente 10% menor em comparação com a estratégia *Fanout*.

Tempo de Detecção e Recuperação

Estes experimentos têm o objetivo de verificar a diferença nos tempos de detecção e recuperação para as duas estratégias. Tempo de detecção é o tempo médio entre a falha de um *peer* e sua suspeita por outro *peer*. Tempo de recuperação é o tempo médio entre a recuperação de um *peer* e o tempo que otro *peer* deixa de suspeitar do mesmo. O tempo de recuperação também pode ser visto como o tempo médio que um novo *peer* leva para ser descoberto por outro *peer*.

Os testes realizados são semelhantes aos das seções anteriores e sem perda de mensagens. Em um dado instante, um *peer* interrompe sua execução. Este *peer* volta a executar 10 segundos depois de parar.



(a) (b)

Figura 4. (a) Tempo de detecção de uma falha, para diferentes valores de banda utilizada. (b) Tempo de detecção da recuperação, para diferentes valores de banda utilizada.

A Figura 4(a) mostra o tempo de detecção para diferentes valores de banda utilizada. Pode-se observar pelos resultados uma grande variação nos tempos de detecção. A variação para os valores baixos de uso de banda se devem provavelmente ao maior número de enganos, pois um *peer* tem mais chance de estar suspeito, mesmo que ainda não tenha falhado. O gráfico também mostra que o tempo de detecção para a estratégia de *Fanout* é consideravelmente menor, sendo aproximadamente 20% menor para o valor de banda 50.

Na Figura 4(b), é mostrado o tempo de recuperação para diferentes valores de banda. Neste caso, a estratégia de *Gossip* é superior, possuindo um tempo de recuperação aproximadamente 55% inferior à estratégia de *Fanout* para o valor de banda 50. Esta diferença se deve, provavelmente, à maior frequência de atualizações. Também pode-se observar que o impacto no aumento do uso de banda afeta diretamente o tempo de recuperação.

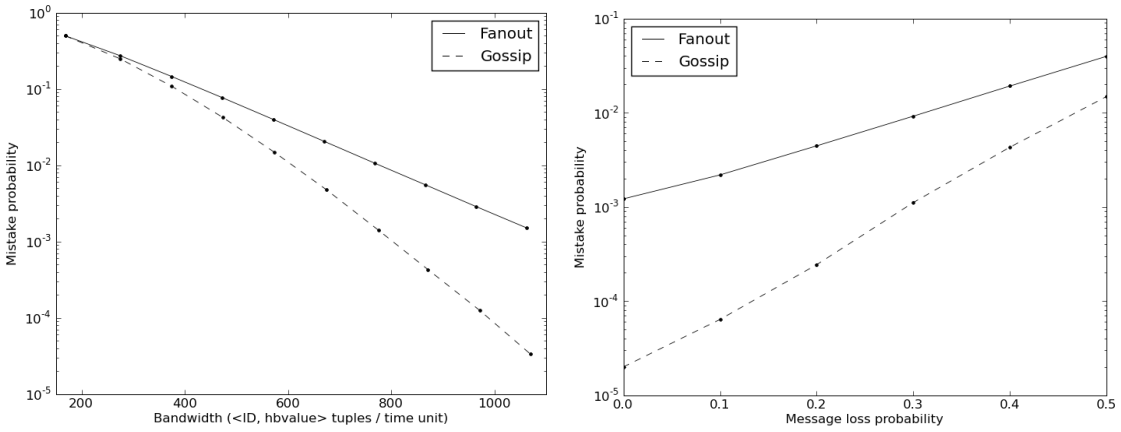
4.3. Resultados da Simulação

Os experimentos de simulação têm como objetivo avaliar o comportamento do algoritmo de detecção para um grupo maior de *peers*, sem a interferência da plataforma JXTA nos resultados.

Os experimentos de simulação foram realizados com um grupo de 200 peers. Alguns parâmetros são mantidos em todos os experimentos, para facilitar a interpretação dos resultados. O $SUSPECT\_TIME$ é igual a 5 unidades de tempo e o $REMOVE\_TIME$ é igual a 20 unidades de tempo. O detector é consultado a cada 0.25 unidades de tempo. A rotina *BroadcastTask* é executada a cada 1 unidade de tempo, sendo o $BCAST\_MAX\_PERIOD$ igual a 20 unidades de tempo e o $BCAST\_FACTOR$ igual a 8.2, fazendo com que um *broadcast* seja feito a cada 10 unidades de tempo, aproximadamente. A estratégia *Gossip* mantém o $FANOUT$ em 1 e varia o $GOSSIP\_INTERVAL$, enquanto a estratégia $FANOUT$ mantém o $GOSSIP\_INTERVAL$ em 2 unidades de tempo e varia o valor $FANOUT$.

Probabilidade de Enganos

Este experimento teve como objetivo avaliar o impacto do aumento da banda utilizada na exatidão do detector. Nenhum *peer* falha durante o experimento, logo, qualquer suspeita é considerada um engano.



(a) (b)

Figura 5. (a) Probabilidade de engano do algoritmo. Experimento realizado com 50% das mensagens sendo descartadas. (b) Probabilidade de engano do algoritmo para diferentes probabilidades de perda de mensagens.

A Figura 5(a) mostra a variação na quantidade de enganos do detector, de acordo com a banda utilizada pelo algoritmo. A perda de mensagens é de 50%. Pode-se observar que o aumento da frequência das mensagens *gossips* tem um impacto muito maior na diminuição dos enganos do detector, tendo, após certo ponto, uma vantagem de mais de uma ordem de magnitude em relação ao aumento no $FANOUT$.

Na Figura 5(b) pode-se observar o impacto da perda de mensagens no número de enganos do detector. A banda utilizada é fixada em aproximadamente 550 ($FANOUT$ igual a 5 e $GOSSIP\_INTERVAL$ de 0.4 unidades de tempo). O gráfico mostra que a estratégia *Gossip* tem uma probabilidade muito menor de cometer enganos, para todos os valores de probabilidade de falha utilizados.

Estes resultados, juntamente com os resultados apresentados para a implementação JXTA, mostram que a estratégia *Gossip* é bastante superior à *Fa-*

nout em relação à exatidão do detector, ou seja, a probabilidade de enganos. Além disso, a diferença entre as duas estratégias se torna ainda maior com o aumento no número de *peers* do grupo.

Eleição

Este experimento tem o objetivo de verificar a viabilidade do uso do detector proposto para a solução do problema da eleição. São utilizados os mesmos valores dos experimentos anteriores para os parâmetros do detector. Cada *peer* considera como líder o *peer* de menor identificador considerado correto pelo seu detector. Desse modo, a cada 1 unidade de tempo aproximadamente, o detector de todos os *peers* são consultados simultaneamente. Para que a eleição tenta sucesso, todas estas consultas devem indicar o mesmo líder. Nenhum *peer* falha durante estes experimentos.

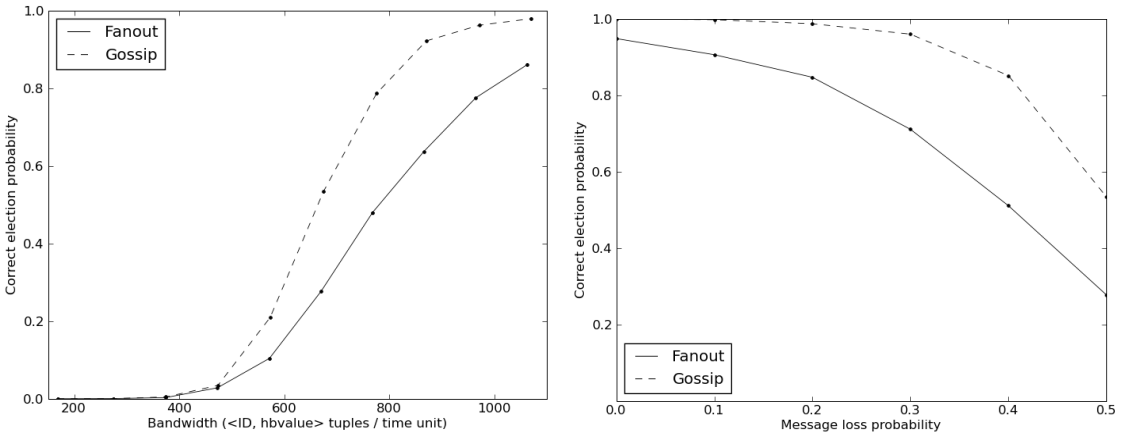


Figura 6. (a) Probabilidade de eleição correta. Experimento com 50% de mensagens perdidas. (b) Probabilidade de eleição correta para diferentes valores de perda de mensagens. A banda utilizada é de aproximadamente 650.

A Figura 6(a) apresenta a probabilidade de uma eleição ser correta para diversos valores de banda utilizada. Mensagens são perdidas com 50% de chance. Este gráfico mostra que a estratégia de *Gossip* é mais eficaz para a execução do algoritmo de eleição. Alem disso, pode-se verificar que resultados razoáveis somente são obtidos com valores maiores de banda, ou seja, baixa probabilidade de enganos.

Na Figura 6(b) podemos verificar o impacto da perda de mensagens na eleição. A banda utilizada é fixada em aproximadamente 650 ($FANOUT$ igual a 6 e $GOSSIP\_INTERVAL$ de 0.332 unidades de tempo). O gráfico mostra que a perda de mensagens causa grande impacto no resultado da eleição. Pode-se observar também que a eleição utilizando a estratégia *Gossip* se mostra consideravelmente mais resistente a falhas.

Estes resultados mostram que a eleição é probabilística, tendo maior chance de sucesso quanto maior a banda utilizada pelo algoritmo de detecção. Fica claro também que a estratégia *Gossip* obtém melhores resultados que a *Fanout* para um mesmo valor de banda utilizada.

5. Conclusão

Este trabalho apresentou a especificação, implementação e avaliação de um serviço de detecção de falhas baseado em disseminação epidêmica. O serviço foi implementado na plataforma JXTA e, para permitir a avaliação do detector com um número maior de *peers* e sem a interferência do JXTA, foi implementado um simulador com a biblioteca SMPL. *Peers* que implementam o serviço e participam de um grupo de monitoração podem consultar o detector para obter informações sobre o estado dos outros *peers* do grupo.

O serviço de detecção foi avaliado através de experimentos para a plataforma JXTA e simulador. Os resultados obtidos demonstram que o algoritmo de disseminação epidêmica é escalável para o número de *peers* participantes e, utilizando o valor de banda necessário, robusto. Os resultados também mostram que a estratégia de disseminação epidêmica resulta em um número consideravelmente menor de enganos do detector em comparação com uma estratégia equivalente mas baseada em aumento do *fanout*. A diferença entre as estratégias se torna ainda maior com o aumento do número de *peers*. O único resultado desfavorável para a estratégia de disseminação foi relacionado ao tempo de detecção de falhas, em média maior do que o tempo da estratégia de *fanout*.

A decisão pelo uso da plataforma JXTA foi motivada pela disponibilidade de facilidades para o desenvolvimento de aplicações par-a-par, mais especificamente os mecanismos de *relay*, *rendezvous* e *multicast* de mensagens (*propagated pipes*), que deveriam funcionar de maneira transparente. A realidade porém é bastante distinta. Os componentes não funcionaram, impossibilitando testes com máquinas em redes diferentes. A documentação existente era insuficiente ou desatualizada. Uma outra opção seria o uso dos *relays* e *rendezvous* públicos, fornecidos pela comunidade JXTA, mas os mesmos nunca estiveram disponíveis durante a implementação do serviço e realização dos experimentos.

Para trabalhos futuros, alguns caminhos podem ser apontados. O primeiro, seria a execução do serviço JXTA-FD em um número maior de *hosts*, utilizando os mecanismos de *rendezvous* e *relay* quando necessário. Mais testes também poderiam ser feitos para verificar o impacto do JXTA na exatidão do algoritmo de detecção. Outro caminho, seria a implementação do serviço de detecção para outras plataformas e tecnologias. Estas implementações poderiam então ser comparadas entre si e com a implementação original na plataforma JXTA. Por fim, algum algoritmo para solução de problema de acordo poderia ser implementado utilizando como base o serviço de detecção. Um exemplo seria o algoritmo Paxos[Lamport 1998] para a resolução do consenso.

Referências

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

Chen, W., Toueg, S., and Aguilera, M. K. (2002). On the quality of service of failure detectors. *IEEE Trans. Comput.*, 51(1):13–32.

Das, A., Gupta, I., and Motivala, A. (2002). Swim: scalable weakly-consistent infection-style process group membership protocol. In *Proc. International Conference on Dependable Systems and Networks DSN 2002*, pages 303–312.

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

Greve, F. G. P. (2005). Protocolos fundamentais para o desenvolvimento de aplicações robustas. *SBRC'05*.

Gupta, I., Birman, K. P., and van Renesse, R. (2002). Fighting fire with fire: using randomized gossip to combat stochastic scalability limits. *Quality and Reliability Engineering International*, 18(3):165–184.

Gupta, I., Chandra, T. D., and Goldszmidt, G. S. (2001). On scalable and efficient distributed failure detectors. In *PODC '01: Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 170–179, New York, NY, USA. ACM.

JXTA (2009). Jxta community website. `https://jxta.dev.java.net/`, acessado em junho de 2009.

Lamport, L. (1998). The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169.

MacDougall, M. H. (1997). *Simulating Computer Systems, Techniques and Tools*. The MIT Press.

Raynal, M. (2005). A short introduction to failure detectors for asynchronous distributed systems. *SIGACT News*, 36(1):53–70.

Turek, J. and Shasha, D. (1992). The many faces of consensus in distributed systems. *Computer*, 25(6):8–17.

van Renesse, R., Minsky, Y., and Hayden, M. (1998). A gossip-style failure detection service. Technical report, Cornell University, Ithaca, NY, USA.

Controle de Admissão para QoS em Sistemas Distribuídos Híbridos, Tolerantes a Falhas

Sérgio Gorender, Raimundo José de Araújo Macêdo, Waltemir Lemos Pacheco Júnior

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

{gorender,macedo}@ufba.br, brwaltemir@gmail.com

Abstract. *Hybrid distributed systems have synchronous and asynchronous processes and communication channels. Depending on the amount of synchronous components in this system, it is possible to solve classical problems of distributed systems, such as consensus, with a higher level of fault tolerance. Hybrid models for fault tolerant distributed systems have been presented with these features. Synchronous communication channels can be obtaines through the use of QoS Architectures. These architectures, while based on different mechanisms, usually show some kind of service that provides a communication isochronous service (synchronous channel). Admission control mechanisms are fundamental for providing isochronous services for new communication channels. Using this mechanism it is possible to ensure that there is no overloading of the network resources reserved for the isochronous classe of service. In this paper we present an admission control module for the QoS Provider, which is a mechanism for managing QoS and is used by models for hybrid distributed systems such as HA and Spa.*

Resumo. *Sistemas distribuídos híbridos são compostos por processos e canais de comunicação que podem ser síncronos ou assíncronos. Dependendo da quantidade de componentes síncronos presente no sistema, é possível resolver problemas clássicos dos sistemas distribuídos, como o consenso, com um maior nível de tolerância a falhas. Modelos para sistemas distribuídos híbridos, tolerantes a falhas têm sido apresentados com estas características. Uma das formas de se obter canais de comunicação síncronos é através do uso de arquiteturas para prover QoS. Estas arquiteturas, embora baseadas em mecanismos diferentes, em geral apresentam alguma classe de serviço que fornece um serviço de comunicação isócrono (síncrono). Para que estes serviços isócronos sejam possíveis, é fundamental o uso de um mecanismo de controle de admissão para novos canais de comunicação, para garantir não haver sobrecarga dos recursos de rede utilizados para prover o serviço. Apresentamos neste artigo um módulo de controle de admissão para o QoS Provider, o qual é um mecanismo para gerenciamento de QoS sendo utilizado por modelos para sistemas distribuídos híbridos como o HA e o Spa.*

Palavras-chave: QoS, Controle de Admissão, modelos para sistemas distribuídos híbridos, tolerância a falhas, detecção de defeitos.

1. Introdução

Sistemas distribuídos híbridos são compostos por componentes (processos e canais de comunicação) que podem apresentar um comportamento síncrono ou assíncrono. Não são portanto sistemas síncronos, mas possuem componentes síncronos em execução. Nestes sistemas é possível executar protocolos distribuídos, com o consenso, tolerando falhas, proporcionalmente ao número de componentes síncronos existentes no sistema, sendo o próprio consenso um importante bloco de construção para a criação de sistemas distribuídos tolerantes a falhas.

Nos sistemas híbridos é possível tirar proveito do nível de sincronismo existente para resolver problemas não solucionados nos sistemas assíncronos, tolerando falhas. Os modelos HA [Gorender et al. 2007] e Spa [Macêdo and Gorender 2009] para sistemas distribuídos híbridos apresentam propriedades e características utilizadas na solução destes problemas.

Ambientes de execução híbridos se tornam comuns nos dias atuais, e existem diversas formas de se implementar processos e canais de comunicação síncronos e assíncronos. É possível executar ações dos processos com limites de tempo garantidos através do uso de sistemas operacionais de tempo real, como por exemplo, o Xenomai ou o RTLinux. Também podemos obter a execução síncrona de processos utilizando computadores dedicados, dimensionados para executar os processos com limites de tempo garantidos. Também é possível obter canais de comunicação síncronos com o uso de redes de controle dedicadas, dimensionadas para a comunicação a ser efetuada. Assim como podemos obter um serviço de comunicação síncrono com o uso de Qualidade de Serviço (QoS).

Qualidade de Serviço diz respeito à possibilidade de se reservar recursos de rede (largura de banda e memória nos roteadores) para alguns fluxos de comunicação (canais de comunicação) e de se priorizar estes fluxos, no encaminhamento das mensagens nos roteadores. A reserva e priorização podem ser efetuadas por fluxos de comunicação ou por agrupamentos de fluxos de comunicação (classes). A arquitetura DiffServ, desenvolvida pelo IETF, provê reserva de recursos e prioridade para classes de encaminhamento de pacotes, sendo que os fluxos de comunicação são atribuídos a estas diferentes classes. Estas classes são configuradas nos roteadores, os quais passam a prover o serviço especificado.

Para garantir o fornecimento de um serviço síncrono, é necessário que a quantidade de fluxos de comunicação alocados à classe de serviço não gere uma sobrecarga nos recursos reservados para esta classe, garantindo que, no pior caso, todos os pacotes recebidos pelos roteadores e atribuídos a esta classe serão encaminhados, não havendo perdas de pacotes. Para tal, torna-se necessário a utilização de um mecanismo de Controle de Admissão, o qual irá verificar a disponibilidade de recursos nos roteadores, e só admitirá um novo fluxo de comunicação para uma classe de serviço, se esta classe ainda tiver recursos disponíveis em quantidade suficiente para tal.

Neste artigo apresentamos a implementação de um mecanismo de Controle de Admissão para o QoS Provider [Gorender et al. 2004], o qual é um mecanismo para a criação de canais de comunicação e gerenciamento de informações sobre os serviços de comunicação fornecidos aos canais criados. O QoS Provider foi desenvolvido para prover

informações a sistemas distribuídos híbridos, sobre o estado dos componentes do sistema, entre síncronos e assíncronos (timely e untimely), assim como, permitir a comunicação destes sistemas com mecanismo do ambiente de execução, como arquiteturas para prover QoS e sistemas operacionais de tempo real. Um protótipo simplificado deste mecanismo foi apresentado em [Gorender et al. 2004], porém sem um módulo de controle de admissão.

Estes canais de comunicação síncronos, fornecidos a partir de um controle de admissão, são utilizados por detectores de defeitos para a execução de detecções perfeitas. Desta forma é possivel implementar, nestes ambientes híbridos, detectores de defeitos híbridos, que realizam detecções não confiáveis (suspeitas), e detecções confiáveis, assim como um detector de defeitos perfeito, dependendo da quantidade e organização dos componentes híbridos do sistema [Macêdo and Gorender 2009].

Este artigo está organizado da seguinte forma: a seção a seguir apresenta algumas características dos modelos para sistemas distribuídos híbridos, a seção 3 apresenta os conceitos básicos sobre Qualidade de Serviço, e a relevância e estratégias adotadas para módulos de Controle de Admissão, a seção seguinte, 4, apresenta a estrutura geral do mecanismo de controle de admissão desenvolvido para o QoS Provider, a seção 5 apresenta diversos aspectos da implementação do mecanismos de controle de admissão, a seção 6 mostra os resultados de testes realizados com canais de comunicação com e sem QoS, admitidos com o uso do controle de admissão, e a seção 7 apresenta conclusões a este trabalho.

2. Sistemas Distribuídos Híbridos

Modelos para sistemas distribuídos são definidos a partir de suas propriedades e características. O modelo síncrono apresenta restrições temporais para a execução de ações dos processos e para a transferência de mensagens entre estes processos, além de relógios com desvios limitados. O modelo assíncrono não apresenta restrições temporais. Diversos modelos ditos parcialmente síncronos têm sido propostos, caracterizados por inserir algum nível de sincronismo ao sistema assíncrono. Os sistemas híbridos possuem componentes (processos e canais de comunicação) síncronos e assíncronos.

Um modelo para sistemas distribuídos híbrido é composto por processos e canais de comunicação que podem ser síncronos ou assíncronos. O modelo HA considera que todos os processos são síncronos, mas os canais de comunicação podem ser síncronos ou assíncronos [Gorender et al. 2007]. Também assume que os canais de comunicação podem alterar seu estado entre síncrono e assíncrono. Para este modelo foi desenvolvido um detector de defeitos também híbrido, que realiza suspeitas e notificações de processos, e que se adapta a alterações no estado dos canais de comunicação, entre síncrono e assíncrono. Já o modelo Spa assume que tanto processos quanto canais de comunicação podem ser síncronos e assíncronos, mas que este estado não se modifica [Macêdo and Gorender 2009]. Neste modelo, os processos síncronos interligados por canais de comunicação síncronos são agrupados em partições síncronas. Nestas partições podemos realizar detecção de defeitos perfeita. Se todos os processos são síncronos e pertencem a alguma partição síncrona, implementamos no sistema um detetor de defeitos perfeito (P).

Para obter estes modelos, necessitamos de mecanismos que permitam obter

canais de comunicação com características isócronas (síncrono), assim como processos que executem tarefas síncronas. Para obter os canais de comunicação utilizamos uma arquitetura para prover Qualidade de Serviços em redes de computadores. Para obter processos síncronos utilizamos um sistema operacional de tempo real.

3. QoS e Controle de Admissão

Qualidade de Serviço (QoS), de uma forma geral, diz respeito ao modo como um serviço (execução de tarefas ou comunicação, por exemplo) pode ser oferecido a uma aplicação com alta eficiência ou, simplesmente, sem que haja perdas em seu desempenho. O serviço é especificado por um conjunto de requisitos. Estes requisitos podem ser qualificados, quantificados e utilizados como parâmetros para caracterizar o tipo de qualidade de serviço oferecido. Atualmente, a qualidade de serviço pode ser aplicada a vários níveis arquiteturais, tais como a nível de sistemas operacionais, subsistemas de comunicação e na comunicação de dados [Aurrecoechea et al. 1998]. O conceito de QoS é amplamente utilizado na área de redes para referenciar a qualidade de serviço aplicada a um determinado serviço ou fluxo de dados (WANG, 2001). Em uma rede com QoS, alguns fluxos de comunicação podem ser privilegiados, ou seja, parte dos recursos da rede podem ser reservados para atender às necessidades especiais destes em detrimento de outros. Estes fluxos de comunicação poderão obter reserva de largura de banda e *buffer* nos roteadores, assim como maior prioridade para o encaminhamento de seus pacotes de dados. Existem diversas arquiteturas desenvolvidas para prover QoS a redes de comunicação, tais como: Quartz [Siqueira and Cahill 2000], Omega [Nahrstedt and Smith 1995], QoS-A [Campbell et al. 1994], IntServ [Braden et al. 1994] e DiffServ [Blake et al. 1998]. As arquiteturas IntServ e DiffServ foram padronizadas pelo IETF (*Internet Engeneering Task Force*)

Estas arquiteturas gerenciam os recursos das redes, e fornecem serviços com qualidade para fluxos de comunicação. Para que um melhor serviço de comunicação seja provido a um novo fluxo de comunicação é necessário que a arquitetura verifique a disponibilidade de recursos na rede, e realize uma reserva de recursos, em quantidade suficiente, para prover o serviço requisitado. A verificação da disponibilidade dos recursos é feita por um mecanismo de controle de admissão. Este mecanismo verifica a disponibilidade de recursos na rede, em toda a rota a ser percorrida pelo fluxo de comunicação, e só admite o novo fluxo, com o compromisso de prover a Qualidade de Serviço solicitada, se existirem recursos suficientes na rede. No caso da admissão do fluxo, os recursos são reservados, não estando mais disponíveis para a admissão de um outro fluxo de comunicação.

3.1. Controle de Admissão em um Domínio DIFFSERV

Um domínio de rede que provê QoS utilizando a arquitetura DiffServ é chamado de Domínio DiffServ. Esta arquitetura provê QoS classificando os diversos fluxos de comunicação em diferentes níveis de serviço. O IETF padronizou três classes de serviço para o DiffServ: Serviço Melhor Esforço (padrão da Internet), Serviço Assegurado (provê níveis diferentes de prioridade, e de probabilidade em não haver perdas de pacotes) e o Serviço Expresso (garante não haver perdas de pacotes, e um atraso limitado na transferência destes pacotes). Enquanto o Serviço Melhor Esforço se caracteriza por uma comunicação não isócrona (assíncrona), o Serviço Expresso provê uma comunicação

isócrona (síncrona). Uma vez que um novo fluxo de comunicação é admitido para uma classe, os pacotes gerados para este fluxo são marcados, na origem (ou pelo primeiro roteador do domínio DiffServ, chamado roteador de borda), com um código (*DSCP - DiffServ Codepoint*) que identifica para os roteadores a classe de serviço à qual o fluxo de pacotes foi admitido.

Para garantir que os fluxos atribuídos ao Serviço Expresso recebam este serviço é fundamental a utilização de um serviço de controle de admissão. Nesta arquitetura, o controle de admissão irá garantir que não haverá sobrecarga sobre os recursos alocados à classe Serviço Expresso. É garantido que todo pacote recebido é encaminhado.

Para implementar o Controle de Admissão, existem basicamente duas abordagens: a distribuída e a centralizada. No Controle de Admissão distribuído, cada roteador pertencente a um domínio deve ter a capacidade de negociar e administrar os seus recursos, ou seja, eles são responsáveis tanto pela admissão dos fluxos quanto pela alocação dos recursos requisitados. Para executar este mecanismo, a aplicação solicita a QoS desejada diretamente aos roteadores, os quais trocam mensagens entre sí. Para isto, utilizam um protocolo de sinalização como, por exemplo, o protocolo RSVP (Resource Reservation Protocol)[Braden et al. 1997]. Os roteadores irão receber a solicitação de requisitos de QoS necessários para a aplicação e, com base em políticas de admissão, irão aceitar ou não a admissão do novo fluxo, determinando assim a melhor forma de alocação dos recursos solicitados. No Controle de Admissão centralizado, um agente central é responsável por administrar os recursos de um domínio e admitir novos fluxos com base em políticas de aceitação. Este agente recebe os pedidos de QoS das aplicações e, com base em informações obtidas da rede, pode decidir se é possível admitir um novo fluxo ou não. Para tal, ele deve ter a capacidade de se comunicar com os roteadores da rede, colher informações e armazená-las. Em algumas arquiteturas é proposta a existência de um agente chamado Bandwidth Broker [Nahrstedt and Smith 1995], com esta responsabilidade.

A estratégia adotada para verificar a possibilidade de se admitir um novo fluxo na rede depende dos requisitos da aplicação, e do nível de serviço solicitado. Se existem requisitos, por parte da aplicação, para um alto nível de QoS, a rede precisará fornecer garantias rígidas em relação ao serviço oferecido ao novo fluxo de comunicação admitido, e neste contexto, é mais adequado o uso de controle de admissão baseado na disponibilidade de recursos. Se a aplicação não requerer um alto nível de serviço, não sendo portanto necessário que a rede forneça garantias rígidas com relação ao serviço fornecido, o controle de admissão pode ser probabilístico, sendo Baseado em Medidas.

4. Controle de Admissão no QoS Provider

O QoS Provider (QoSP) é um mecanismo desenvolvido para gerenciar as informações mantidas por arquiteturas para prover QoS, provendo canais de comunicação com serviços síncrono e assíncrono, e fornecendo aos sistemas distribuídos informação a cerca do serviço provido a cada canal. Além disto, o QoSP fornece controle de admissão a um domínio de rede executando a arquitetura DiffServ, focando nos serviços Expresso e Melhor Esforço (serviço síncrono e assíncrono). O objetivo imediato do desenvolvimento deste mecanismo é a construção de ambientes de execução híbridos, capazes de fornecer componentes (processos e canais de comunicação) síncronos e

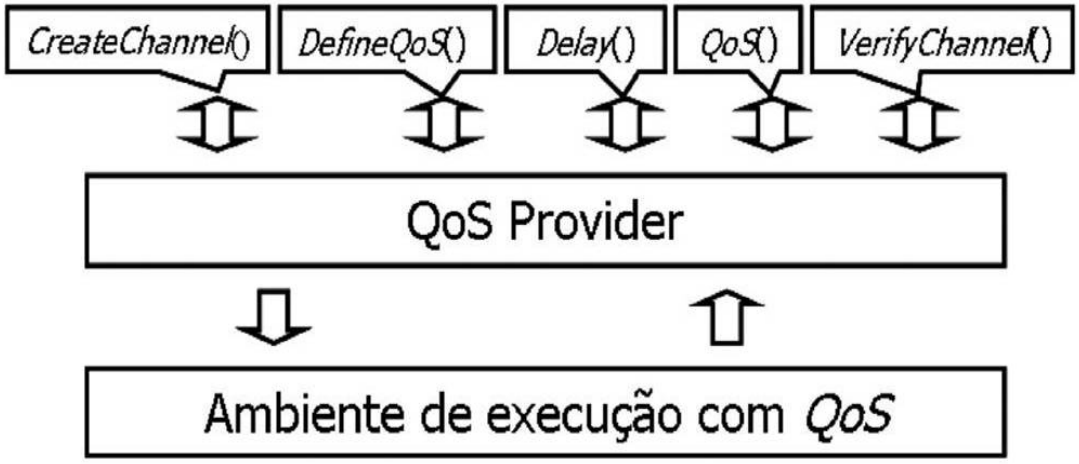


Figura 1. Interface do QoS Provider.

assíncronos aos sistemas distribuídos e o fornecimento de informações sobre a QoS provida a cada canal, informação utilizada por detectores de defeitos híbridos ao realizar detecções, as quais podem ser confiáveis se for estiver sendo utilizado um canal síncrono. Modelos para sistemas distribuídos como o HA [Gorender et al. 2007] e o Spa [Macêdo and Gorender 2009] são modelos híbridos, baseados na existência de canais de comunicação síncronos e assíncronos (no caso do modelo HA), e canais e processos síncronos e assíncronos (no caso do modelo Spa). Para ambos os modelos, uma das formas de se fornecer canais síncronos é através do uso de arquiteturas para prover QoS.

O módulo de controle de admissão desenvolvido para o QoS Provider executa interagindo com uma implementação da arquitetura DiffServ disponível em roteadores Cisco. Este módulo foi baseado na abordagem centralizada, e na estratégia de controle de admissão baseada em disponibilidade de recursos. A função do QoS Provider que representa o controle de admissão é *defineQoS()*, como apresentada na Figura 1. As demais funções do QoSP são responsáveis por criar canais de comunicação (CreateChannel, verificar o atraso máximo para a transferência de mensagens (Delay), verificar se um canal de comunicação é *timely* ou *untimely* (QoS) e verificar se um canal de comunicação remoto apresenta tráfego (VerifyChannel). As justificativas e implementação destas funções está fora do escopo deste trabalho.

O QoSP gerencia e provê dois níveis de serviço, fornecendo canais de comunicação chamados *timely* e *untimely*. Canais *timely* (síncronos) são atribuídos ao Serviço Expresso, definido pela arquitetura DiffServ através do *PHB Expedited Forwarding* [Davie et al. 1999] enquanto o canais *untimely* são providos com o Serviço Melhor Esforço, definido pela arquitetura DiffServ como *PHB Default* [Blake et al. 1998].

Cada *host* do sistema contém um módulo ativo do QoSP, o qual funciona como um servidor local para as aplicações que estão rodando no mesmo *host*. Os serviços são requisitados através do envio de mensagens de solicitações, utilizando para isto as funções fornecidas por uma API cliente, a qual tem uma interface de comunicação para a troca de mensagens com o QoSP. O QoSP, localizado no host dos processos p_x e p_y , é definido, respectivamente, como QoSP$_x$ e QoSP$_y$.

O QoSP Bandwidth Broker (QoSPBB) foi desenvolvido baseado no Bandwidth Broker [Nahrstedt and Smith 1995], para dar suporte ao Controle de Admissão do QoSP.

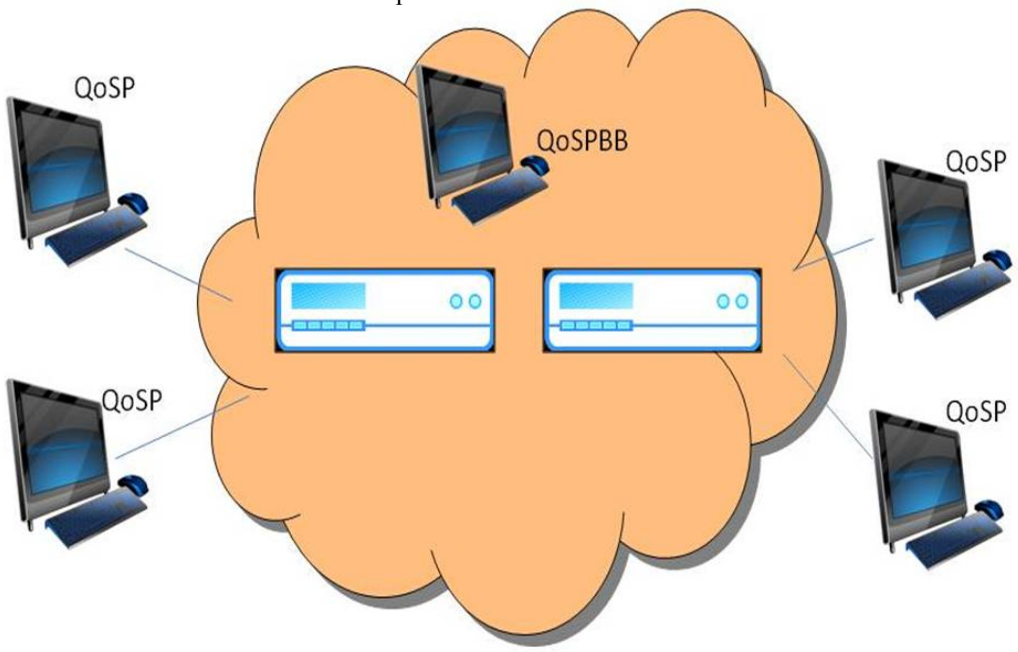


Figura 2. Domínio QoS Provider

O QoSPBB executa em um servidor ligado a algum roteador do domínio de rede, estabelecendo comunicação com todos os roteadores do domínio, e com os módulos QoSP nos *hosts*. A comunicação entre os módulos QoSP cliente e o QoSPBB se dá através de um protocolo de comunicação desenvolvido para tal. O QoSPBB foi projetado para aceitar apenas as requisições dos QoSPs distribuídos pelo domínio de rede, utilizando para isto um mecanismo de autenticação.

Na inicialização de um módulo QoSP, a mensagem $QOSP\_REGISTER$ é enviada ao QoSPBB, com o intuito de registrar o módulo QoSP como cliente do QoSPBB. O controle de admissão é iniciado por um dos processos da aplicação, através de uma solicitação ao módulo QoSP existente em seu *host*. O módulo QoSP estabelece uma comunicação com o QoSPBB e com o módulo QoSP no *host* destino. O resultado da verificação de disponibilidade de recursos para a classe Serviço Expresso nos roteadores do domínio, caso positiva, é armazenada pelo QoSPBB como um Acordo de Nível de Serviço (*Service Level Agreement - SLA*).

5. Implementação do Controle de Admissão no QoSP

O QoSP foi implementado em C++, sobre uma rede de computadores composta por desktops com o sistema operacional linux, e roteadores CISCO 871, com o sistema *IOS() Advanced IP Service*, o qual dá suporte à arquitetura DiffServ. Descrevemos a seguir a API do QoSP, o protocolo de comunicação entre as aplicações e os módulos QoSP, o protocolo de comunicação entre os módulos QoSP eo QoSPBB e o mecanismo de comunicação entre o QoSPBB e os roteadores. A API e as mensagens descritas consideram também funções para criar (*CreateChannel()*) e excluir canais de comunicação. As funções *Delay()*, *QoS()* e *VerifyChannel()*, apresentadas na figura 1 não são descritas neste trabalho.

5.1. A API do QOSP

Os serviços do QoSP são solicitados pelas aplicações, utilizando para isso as funções de uma API. As solicitações são feitas para o módulo do QoSP, o qual está localizado no host

onde as aplicações solicitantes estão rodando. A API funciona como um programa cliente, o qual deve ser anexado às aplicações através do arquivo cabeçalho $api\_qosprovider.h$. Quando uma função da API é utilizada para requisitar um serviço, isto implicará no envio de uma mensagem ao QoSP, conforme o serviço solicitado. A API contém uma função para cada serviço do QoSP. As funções relacionadas com o controle de admissão são:

- $qosp\_init\_api()$: Esta função é utilizada para inicializar as estruturas de comunicação com o módulo do QoSP. Inicializa também a estrutura *tableSockChannel*. Esta função não tem parâmetros de entrada e não solicita serviços ao QoSP.
- $qosp\_createChannel()$: Esta função é utilizada quando a aplicação solicita o registro da criação de um canal de comunicação. Ela recebe os endereços IP e as portas que serão utilizadas pelos processos para a comunicação. Esta função envia a mensagem $CHANNEL\_REQUEST$ para o módulo do QoSP e recebe a resposta através da mensagem $CHANNEL\_REPLY$.
- $qosp\_deletChannel()$: Esta função é solicitada quando uma aplicação deseja finalizar um canal de comunicação. Ela recebe como entrada um valor que identifica o canal a ser finalizado, e envia a mensagem $CHANNEL\_DELET$ ao módulo do QoSP.
- $qosp\_defineQos()$: A aplicação utiliza esta função para solicitar a alteração da QoS provida ao seu canal de comunicação. Isto implica na mudança do tipo de canal utilizado, ou seja, de *timely* para *untimely* ou vice-versa. Os parâmetros de entrada são o identificador do canal de comunicação e uma estrutura que foi definida para conter os parâmetros de QoS que a aplicação necessita. Além dos parâmetros de QoS, esta estrutura contém um valor inteiro, que representa o tipo de canal solicitado (*timely* ou *untimely*). Os parâmetros de QoS só serão enviados para o módulo QoSP através da mensagem $DEFINEQOS\_REQUEST$, caso a solicitação seja de *untimely* para *timely*.

A estrutura *tableSockChannel* citada na função $qosp\_init\_api()$ é um vetor onde são armazenadas as identificações dos canais, ou seja, para cada valor de *socket* criado existirá um *idChannel* equivalente. O *socket* é um valor inteiro vinculado a uma porta de comunicação, o qual identifica a porta a ser utilizada no momento do envio da mensagem. Sendo assim, este identifica o canal para a aplicação. Já o *idChannel* é um valor utilizado pelo módulo QoSP como identificador de um canal. Logo, a estrutura *tableSockChannel* foi criada para relacionar ambos os identificadores.

5.2. Protocolo de comunicação de aplicações com o módulo QoSP

As mensagens que a aplicação envia ao QoSP através da API são constituídas de dois campos padronizados. O primeiro campo da mensagem é o identificador (*Id*) do tipo de mensagem enviada, ou seja, o módulo QoSP vai utilizar este *Id* para identificar o tipo de solicitação, como por exemplo, a criação de um canal, a negociação de QoS, etc. O segundo campo da mensagem (*Information*) contém informações que serão relevantes para o serviço solicitado. Segue abaixo uma descrição das mensagens de solicitação de serviços enviadas a um módulo QoSP:

- $CHANNEL\_REQUEST$: Mensagem enviada pela aplicação ao módulo do QoSP para requisitar o registro de um canal de comunicação. Para enviar esta

mensagem, a aplicação utiliza a função $qosp\_CreateChannel()$ da API. Esta mensagem é formada pelas seguintes informações: *Id* da mensagem, endereço IP do processo p_x, porta do processo p_x, endereço *IP* do processo p_y e porta do processo p_y.

- $CHANNEL\_REPLY$: Esta mensagem é enviada em resposta à solicitação do registro de criação de canal, ou seja, em resposta à mensagem $CHANNEL\_REQUEST$. Esta mensagem contém o identificador do canal criado, o qual será armazenado na estrutura *tableSockChannel* junto com o valor do *socket* equivalente para o canal.
- $CHANNEL\_DELET$: Mensagem enviada ao módulo QoSP para solicitar a exclusão de um canal. Além do *Id* equivalente à solicitação, esta mensagem leva como informação o *idChannel* do canal a ser excluído. Ao receber esta solicitação, o módulo QoSP excluirá as informações do canal armazenado e, caso o canal seja *timely*, os recursos alocados para este serão liberados.
- $DEFINEQOS\_REQUEST$: Esta mensagem é enviada ao QoSP para solicitar a negociação de QoS para um determinado canal de comunicação. Esta mensagem contém, além do *Id* equivalente à solicitação, o *idChannel* do canal e os parâmetros de QoS a serem negociados. Entretanto, caso a negociação seja de *timely* para *untimely*, os parâmetros de QoS não irão compor a mensagem.
- $DEFINEQOS\_REPLY$: Esta mensagem é enviada em resposta à solicitação de negociação de QoS, ou seja, em resposta à mensagem $DEFINEQOS\_REQUEST$. O conteúdo da mensagem identificará se a solicitação foi atendida ou não.

5.3. Protocolo de comunicação entre os módulos QoSP e o QoSPBB

Um módulo QoSP comunica-se com outros módulos QoSP distribuídos na rede e com o QoSPBB, com o intuito de executar seus serviços. Foi criado um protocolo de comunicação para a execução das diversas funcionalidades implementadas, para a realização do controle de admissão.

- $CHANNEL\_REGISTER$: Esta mensagem é enviada de um módulo QoSP para outro, quando um canal é criado entre dois processos. Por exemplo, quando o processo p_x solicita a criação de um canal ao $QoSP_x$, uma mensagem $CHANNEL\_REGISTER$ é enviada ao $QoSP_y$ para que o mesmo registre o canal. Além do *Id*, que identifica o tipo de mensagem, a mensagem também contém as informações referentes ao canal.
- $CHANGE\_QOS$: Esta mensagem é enviada de um módulo QoSP para outro, quando a QoS fornecida a um canal é alterada, o que significa alterar entre os dois tipos de canais possíveis (*timely* e *untimely*). Esta mensagem contém o identificador da mensagem, o identificador do canal, o identificador do acordo SLA registrado no QoSPBB e a nova QoS do canal.
- $CLOSE\_CHANNEL$: Esta mensagem é enviada de um módulo QoSP para outro, quando um canal é encerrado. Quando um processo p_x solicitar o encerramento de um canal ligando os processo p_x e p_y, a mensagem $CLOSE\_CHANNEL$ é enviada ao $QoSP_y$ para que o mesmo também exclua o canal do seu banco de dados. Esta mensagem contém como informação, além do *Id* da mensagem, o identificador do canal a ser excluído.

- $QOSP\_REGISTER$: Esta mensagem é enviada de um módulo QoSP para o QoSPBB. Esta mensagem tem a finalidade de registrar o módulo QoSP no QoSPBB, sendo que este registro é utilizado para autenticar o cliente no momento em que o mesmo solicitar algum serviço ao QoSPBB. Esta mensagem contém o identificador da mensagem e uma senha (*password*) gerada pelo cliente (o módulo QoSP).
- $REGISTER\_REPLY$: Esta mensagem é enviada do QoSPBB para um módulo QoSP, em resposta à mensagem $QOSP\_REGISTER$. A finalidade desta mensagem, além de confirmar o registro do cliente, é de verificar se o QoSPBB está ativo.
- $QOS\_REQUEST$: Esta mensagem é enviada de um módulo QoSP para o QoSPBB quando é feita uma solicitação de reserva de recursos para um canal. Esta mensagem contém como informação, além de seu *Id*, os parâmetros de QoS solicitados pela aplicação, a senha do módulo QoSP e os roteadores que constituem o canal.
- QOS_REPLY: Esta mensagem é enviada do QoSPBB para um módulo QoSP, em resposta à mensagem $QOS\_REQUEST$. Esta contém como informação o resultado da negociação e o identificador do acordo SLA armazenado no QoSPBB.
- QOS_LET: Esta mensagem é enviada de um módulo QoSP para o QoSPBB quando um canal é alterado de *timely* para *untimely*, e quando um canal é encerrado. Esta mensagem tem a finalidade de autorizar a liberação dos recursos que estavam reservados para o canal. Além do *Id* que identifica a mensagem, ela contém o identificador do acordo SLA do canal e a senha do QoSP.

5.4. Comunicação entre o QoSPBB e os roteadores

O QoSPBB precisa comunicar-se com os roteadores do domínio, com o intuito de colher as informações de que necessita. Para esta finalidade, foi utilizado o protocolo SNMP (Simple Network Management Protocol) [Case et al. 1990], o qual é muito utilizado no contexto de gerenciamento dos dispositivos de rede (roteadores, switches, etc). O funcionamento do SNMP é baseado em uma comunicação entre o agente e o gerente. Os agentes são elementos de software instalados nos equipamentos da rede, com a finalidade de colher informações sobre os dispositivos e enviá-las ao gerente, o qual utilizará estas informações para gerenciar os dispositivos.

As informações dos dispositivos, colhidas pelo agente, estão disponíveis em variáveis. Estas variáveis estão estruturadas hierarquicamente em um formato de árvore, sendo esta estrutura conhecida como MIB (Management Information Base). Para obter informações sobre a QoS fornecida pelo roteador foi utilizada a MIB CISCO-CLASS-BASED-QOS-MIB, fornecida pela Cisco para a comunicação com seus roteadores. Foi utilizado um roteador Cisco 871, com o sistema Cisco IOS.

6. Resultados

O ambiente utilizado para a realização de testes foi composto por um roteador CISCO 871, provido com o sistema IOS Security Bundle with Advanced IP Services, o qual dá suporte à arquitetura DiffServ, e três computadores (hosts) configurados com o sistema operacional Debian Gnu/Linux Lenny. Optamos por não utilizar um SO de tempo real, uma vez que, para testar o mecanismo de admissão, não utilizariamos tarefas

de tempo real. Neste ambiente, foram criadas três redes locais, Rede1(192.168.1.0), Rede2(192.168.2.0) e Rede3(192.168.3.0), onde cada host foi configurado em uma destas redes. O *host* C (Rede 3) foi utilizado para rodar o QOSPBB, e cada um dos outros dois hosts executam um módulo do QoSP e uma ou mais aplicações de teste. O ambiente é formado por um domínio DiffServ, o qual interliga as três redes locais. Este domínio é representado apenas por um roteador de núcleo, que tem o papel de encaminhar os pacotes conforme a classe de serviço à qual eles pertencem. O papel de marcar os pacotes fica como atribuição do QoSP ativo no host de origem do fluxo.

O roteador foi configurado com duas classes de serviços: o Serviço Expresso para os canais *timely*, e o de Melhor Esforço(*best-effort*) para os canais *untimely*. É importante salientar que as mensagens trocadas, entre os módulos QoSP e entre estes módulo e o QoSPBB, utilizam canais *timely*.

Uma aplicação simples, do tipo cliente/servidor, foi implementada para testar tanto os módulos desenvolvidos, o QoSP e o QoSPBB, como os protocolos de comunicação criados. A aplicação de teste utiliza as funções da API do QoSP descritas, para solicitar serviços ao QoSP. O objetivo principal desta aplicação é de testar a integração e comunicação entre os componentes, ou seja, entre a aplicação com o módulo QoSP, entre módulos QoSP e de módulos QoSP com o QoSPBB. A aplicação de teste apresenta um menu de opções, onde o usuário pode optar por criar um ou mais canais de comunicação, utilizando para isto a função $qosp\_createChannel()$. Os canais criados são inicialmente *untimely*. Após a criação, o usuário pode solicitar uma QoS para um determinado canal, através da função $qosp\_defineQos()$, sendo que, os requisitos de QoS desejados são passados como parâmetro, para que sejam negociados. Antes de utilizar as funções $qosp\_createChannel()$ e $qosp\_defineQos()$, responsáveis por criar um canal e negociar uma QoS, respectivamente, as aplicações devem utilizar a função $qosp\_init_api()$. Esta função tem o objetivo de inicializar a comunicação com o módulo do QoSP ativo no *Host*.

Foi utilizado um programa chamado Wireshark para analisar o conteúdo das mensagens trocadas entre os módulos desenvolvidos, validando assim os tipos de mensagens trocadas.O Wireshark é um programa que possibilita capturar os pacotes que chegam ou saem de um *Host*, sendo possível ver o cabeçalho e o conteúdo dos pacotes.

Muitos testes foram realizados utilizando a aplicação de teste. Vários canais *untimely* foram criados e admitidos como canais *timely* em um domínio DiffServ, enquanto existiam recursos para admiti-los. Realizamos diversos testes utilizando os canais criados como *timely* e *untimely*, para verificar e validar o controle de admissão implementado. Os resultados que serão mostrados a seguir, referem-se ao RTT (*Round Trip Time*) das mensagens trocadas, ou seja, o tempo que um pacote demora para ir de uma origem até um destino e retornar, em milissegundos, e às perdas de pacotes. Pacotes perdidos não são retransmitidos. Estes valores foram obtidos a partir de quatro fluxos de dados (F1, F2, F3 e F4), sendo que, para cada fluxo de dados, existe uma aplicação de teste rodando, localizada no *Host A*. Os fluxos gerados pela aplicação de teste correspondem ao envio de 10.000 pacotes, com uma taxa de 3 Kbits/s. O canal utilizado para enviar os pacotes pode ser *timely* ou *untimely*. Em cada experimento, foi calculada a média aritmética do RTT dos pacotes transmitidos, em cada canal, assim como o seu desvio padrão.

O DSCP utilizado para o Serviço Expresso foi o recomendado pela IETF, ou seja, o valor 46. Qualquer outro valor de DSCP é considerado pertencente à classe Serviço Melhor Esforço, o qual representa o serviço oferecido pelos canais *untimely*. As larguras de banda configuradas no roteador para as classes de serviço foram: 10 Kbits para os canais *timely* e 10 Kbits para os canais *untimely*. A associação da largura de banda configurada para as classes com a taxa de transferência dos canais permitiu exaurir os recursos reservados, testando assim o mecanismo de admissão.

A tabela 1 mostra quatro fluxos gerados. Os três primeiros foram admitidos e estão utilizando canais *timely*. O fluxo F4 está utilizando um canal *untimely*, pois não pôde ser admitido em decorrência da falta de recursos. Estes fluxos foram gerados do *Host A* para o *Host B*. Neste teste, não foi gerada sobrecarga dos canais. Pela tabela, pode-se perceber que os tempos médios de RTT dos pacotes, que estão utilizando tanto os canais *timely* quanto o canal *untimely*, tiveram valores muito próximos, e não houve perdas de pacotes para nenhum dos fluxos. Já a tabela 2, mostra os mesmos fluxos da tabela anterior, porém com a utilização do programa *ping* do Linux para gerar carga no sistema. Sendo assim, a tabela 2 mostra que o fluxo F4, que utilizou um canal *untimely*, apresentou $4,45\%$ de perdas de pacotes. Isto se deve à sobrecarga no sistema. Os tempos de RTT continuaram próximos para os dois tipos de canais.

Fluxo	Média	Desvio Padrão	Perdas($\%$)
F1	1,651	1,99	0,000
F2	1,209	1,75	0,000
F3	1,144	0,87	0,000
F4	1,343	1,04	0,000

Tabela 1. Canais sem carga

Fluxo	Média	Desvio Padrão	Perdas($\%$)
F1	1,356	1,27	0,00
F2	1,069	1,00	0,00
F3	1,309	1,20	0,00
F4	1,155	0,99	4,45

Tabela 2. Canais com carga

A tabela 3 apresenta um ambiente sem Controle de Admissão, onde os quatro fluxos gerados estão utilizando os recursos existentes da rede. Para este experimento, também foi utilizado o ping para gerar carga no sistema. Podem-se observar perdas de pacotes para os quatro fluxos de dados. Isso ocorreu devido à falta de recursos para atender completamente a todos os fluxos. Já a tabela 4 mostra um ambiente com Controle de Admissão, onde os fluxos F3 e F4 foram admitidos e passaram a utilizar canais *timely*. Pode-se perceber que não houve perdas de pacotes para os fluxos F3 e F4.

Os testes mostram que com o controle de admissão, os fluxos admitidos como *timely*, que são alocados à classe Serviço Expresso, recebem um serviço de fato diferenciado pelo roteador, não apresentando perda de pacotes em suas mensagens. Os fluxos são admitidos para esta classe apenas quando existem recursos suficientes

Fluxo	Média	Desvio Padrão	Perdas(%)
F1	1,486	1,7	11,20
F2	1,228	1,21	22,90
F3	1,470	1,36	15,85
F4	2,906	2,9	18,54

Tabela 3. Canais sem controle de admissão

Fluxo	Média	Desvio Padrão	Perdas(%)
F1	1,144	0,89	3,32
F2	1,083	0,89	2,80
F3	1,323	1,21	0,00
F4	1,220	1,13	0,00

Tabela 4. Fluxos F3 e F4 timely.

para tal. Não havendo perda de pacotes, não haverá a necessidade de retransmissão, gerando atraso na comunicação. Além disto, como a classe Serviço Expresso tem prioridade no encaminhamento de pacotes, o tempo gasto para o encaminhamento destes pacotes será limitado, relativo ao tamanho do *buffer* reservado para a classe. Entretanto, como este *buffer* é grande, o tempo dos pacotes esperando na fila antes de seu encaminhamento variou bastante, proporcional à quantidade de pacotes recebidos e ainda não encaminhados pelo roteador, o que caracteriza o desvio padrão elevado para a média aritmética obtida. No entanto, como o *buffer* tem um tamanho limitado e não há perda de pacotes nesta classe, existe um limite máximo para o atraso no encaminhamento destes pacotes, o qual é razoavelmente superior ao limite mínimo. No caso de canais de comunicação *untimely*, a perda de pacotes implica na necessidade de retransmissão para que as mensagens sejam entregues a seu destino, implicando em um tempo de comunicação não limitado. Os testes mostraram a importância de um Controle de Admissão para criar canais de comunicação com QoS, evidenciando que, em um ambiente sem Controle de Admissão, os recursos podem não ser suficientes para garantir a comunicação através dos canais estabelecidos.

7. Conclusões

Apresentamos neste artigo o módulo de controle de admissão do QoS Provider. Este módulo executa a função *defineQoS()* do QoSP, sendo responsável por verificar a disponibilidade de recursos nos roteadores da rede (largura de banda e memória) para a admissão de novos canais de comunicação a serem providos com Serviço Expresso pela arquitetura DiffServ em execução nestes roteadores. Estes canais apresentarão um comportamento síncrono, com garantias na entrega das mensagens, enquanto não ocorrerem falhas.

O mecanismo de controle de admissão é composto por módulos que são componentes dos módulos QoSP, em execução nos *hosts* dos clientes, e por módulos QoSPBB, que gerenciam a disponibilidade de recursos nos roteadores de um domínio.

Com os testes realizados, verificamos que os canais admitidos para o Serviço Expresso não apresentam perdas de pacotes, tendo todas as suas mensagens entregues ao

seu destino, enquanto que os canais *untimely* (qualquer outro serviço, em geral Serviço Melhor Esforço), apresentam perdas de pacotes, dependendo da sobrecarga gerada na rede.

Este mecanismo é um bloco de construção fundamental para a obtenção de canais de comunicação síncronos fim-a-fim, atuando em conjunto, no QoS Provider, com um mecanismo de admissão de tarefas de tempo real (a partir da utilização de um sistema operacional de tempo real, no caso o Xenomai), e também com a utilização de QoS no acesso do *host* à rede local, com a utilização de placas de rede RtLink com a reserva de tempo de acesso à rede. Utilizando estes canais síncronos é possível executar protocolos para detecção de defeitos, os quais monitoram processos de forma confiável, e no caso de falhas, detectam os processo faltosos de forma confiável.

Referências

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

Blake, S., Black, D., Carlson, M., Davies, E., Wang, Z., and Weiss, W. (1998). An architecture for differentiated services. *RFC 2475*.

Braden, B., Clark, D., and Shenker, S. (1994). Integrated services in the internet architecture: an overview. *RFC 1633*.

Braden, B., Zhang, L., Berson, S., Herzog, S., and Jamin, S. (1997). Resource reservation protocol (rsvp) - version 1 functional specification. *RFC 2205*.

Campbell, A., Coulson, G., and Hutchison, D. (1994). A quality of service architecture. *ACM Computer Communications Review*, 24(2):6–27.

Case, J., Fedor, M., Schoffstall, M., and Davin, J. (1990). A simple network management protocol (snmp). *RFC 1157*.

Davie, B., Charny, A., Bennet, J. C. R., Benson, K., Boudec, J. Y. L., Courtney, W., Davari, S., Firoiu, V., and Stiliadis, D. (1999). An expedited forwarding phb (per hop behavior). *RFC 3246*.

Gorender, S., Macêdo, R. J. A., and Cunha, M. (2004). Implementação e análise de desempenho de um mecanismo adaptativo para tolerância a falhas em sistemas distribuídos com qos. In *Anais do Workshop de Testes e Tolerância a Falhas, V WTF - SBRC2004*, pages 3–14.

Gorender, S., Macêdo, R. J. A., and Raynal, M. (2007). An adaptive programming model for fault-tolerant distributed computing. *IEEE Transactions on Dependable and Secure Computing*, 4(1):18–31.

Macêdo, R. J. A. and Gorender, S. (2009). Perfect failure detection in the partitioned synchronous distributed system model. In *Proceedings of the The Fourth International Conference on Availability, Reliability and Security (ARES 2009), IEEE CS Press*.

Nahrstedt, K. and Smith, J. M. (1995). The qos broker. *IEEE Multimedia*, 2(1):53–67.

Siqueira, F. and Cahill, V. (2000). Quartz: A qos architecture for open systems. In *International Conference on Distributed Computing Systems*, pages 197–204.

XI Workshop de Testes e Tolerância a Falhas

♦

Sessão Técnica 2

Injeção de Falhas

Injeção de falhas para validar aplicações em ambientes móveis

Eduardo Verruck Acker[1], Taisy Silva Weber[1,] Sérgio Luis Cechin[1]

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

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

Abstract. *It is assumed that the Android platform for smartphones will allow porting a large number of applications for these and other mobile devices. These applications, however, should be carefully tested including the test under faults. This paper discusses the strengths and difficulties of working with this new mobile platform, presents the experience of porting a communication fault injector to Android and introduces a project that aims to provide fault injection tools for testing applications in mobile environments considering the specificities of faults that can occur in these environments.*

Resumo. *Presume-se que o Android, a plataforma móvel para smartphones originalmente desenvolvido pela Google vá permitir portar um grande número de aplicações para esses e outros dispositivos móveis. Essas aplicações deverão ser cuidadosamente testadas, inclusive na ocorrência de falhas. Esse artigo discute as facilidades e dificuldades de trabalhar com esse novo ambiente, apresenta o porte de um injetor de falhas de comunicação para o Android e introduz a linha de pesquisa de um novo projeto, visando prover ferramentas de injeção de falhas para teste de aplicações em ambientes móveis, considerando as peculiaridades das falhas de comunicação que ocorrem nestes ambientes.*

1 Introdução

Para testar técnicas empregadas na detecção e correção de falhas em um sistema que deva atender a algum critério de dependabilidade, é necessária a ocorrência efetiva de falhas. Contudo, essa ocorrência é aleatória e incontrolável. Além disso, a ocorrência real de falhas apresenta taxas relativamente baixas e, portanto, inadequadas para o tempo restrito de aplicação dos testes. Como é inviável ficar esperando que uma falha específica ocorra naturalmente, devem ser usadas técnicas que emulem falhas de forma controlável e com taxas adequadas. Uma solução é injeção de falhas. Essa abordagem de validação está bem consolidada na literatura [Arlat 1990]. Existem diversos injetores de falhas reportados [Hsueh 1997], mas muitos apresentam restrições quanto à disponibilidade e ao sistema alvo ao qual o injetor pode ser aplicado.

Sistemas computacionais móveis estão se popularizando rapidamente. O ambiente móvel traz novos e interessantes desafios. As técnicas de tolerância a falhas baseadas em replicação de componentes não podem, muitas vezes, ser diretamente empregadas devido à limitação de recursos do ambiente, como restrições de energia, volume e peso. Ambientes móveis são mais sujeitos a interferências ambientais, o que torna os dispositivos móveis mais susceptíveis a erros. Tais erros devem ser detectados e corrigidos com o mínimo de percepção do usuário [Krishna 1993] e usando o mínimo

de recursos. Técnicas de tolerância a falhas vêm sendo adaptadas ou desenvolvidas para lidar com essas restrições e suas implementações precisam, portanto, ser cuidadosamente testadas. Aplicações que as empregam precisam ser validadas na presença de falhas.

O artigo trata da validação de aplicações para ambientes móveis através do uso de injeção de falhas. O sistema alvo são aplicações desenvolvidas para o Android. O Android é um ambiente aberto desenvolvido para dispositivos móveis, baseado no kernel Linux versão 2.6, e que permite a criação de aplicações por qualquer programador. Prevê-se que o ambiente será incorporado em um grande número de aparelhos e uma quantidade enorme de aplicações estará disponível em pouco tempo. Muitas dessas aplicações, provavelmente, serão oferecidas prometendo garantir requisitos relacionados à dependabilidade [Avizienis 2004] tais como confiabilidade, segurança funcional e disponibilidade.

A linha de investigação do projeto apresentado ao final do artigo visa propor uma ferramenta de injeção de falhas como parte dos recursos necessários a validação, certificação ou benchmark de dependabilidade das aplicações móveis com requisitos de dependabilidade. O projeto sendo proposto tem por objetivo pesquisar modelos de falhas que capturem o comportamento das falhas em ambientes móveis, desenvolver ferramentas de injeção de falhas para emular as falhas do modelo e aplicar essas ferramentas em benchmarks de dependabilidade de aplicações móveis.

Esse artigo mostra como foi realizado o porte de um injetor de falhas de comunicação para o ambiente Android, o primeiro passo já realizado do projeto proposto. Na seção 2 são apresentados os conceitos de injeção de falhas. Na seção 3 discutem-se os sistemas móveis e o Android. Na seção 4 são apresentadas as ferramentas utilizadas para o desenvolvimento do projeto e na seção 5 a sua integração, comentando as dificuldades encontradas nessa etapa. Na seção 6 são apresentados os testes realizados para comprovar a validade do porte realizado. Na seção 7 discute-se a continuidade do trabalho e por fim, na seção 8 encontram-se as conclusões finais do artigo.

2 Injeção de falhas

Injetores de falhas podem ser construídos em hardware ou software. Os injetores de falhas por hardware adicionam, ao sistema alvo sob teste, componentes físicos que servem para emular e monitorar falhas. Nesse método, o injetor e o monitor não usam recursos do sistema sob teste, não afetando assim o seu desempenho. Contudo, há um custo adicional significativo devido aos componentes extras e restrições à quantidade de pontos de inserção e monitoração de falhas. A injeção via software é mais fácil de implementar e é mais flexível do que por hardware, mas apresenta desvantagens como uma possível interferência na carga de trabalho do sistema sob teste e a limitação da injeção apenas a locais acessíveis ao software [Hsueh 1997]. Mas se localizado no kernel do sistema operacional, um injetor de falhas por software apresenta uma grande abrangência de classes de falhas. Ele pode injetar falhas em todos os níveis de abstração superiores àquelas em que se situa e ainda pode operar com baixíssima interferência nesses níveis, aproximando-se das vantagens de um injetor de falhas por hardware, mas sem o ônus do alto custo de implementação.

O foco deste artigo são as falhas que afetam dispositivos móveis. Além de falhas internas ao dispositivo, que comprometem a operação correta de seus componentes de hardware e software, uma importante classe de falhas são as de comunicação. Cristian (1991) definiu um modelo de falhas que se tornou comum para sistemas distribuídos. Esse modelo compreende falhas de colapso, omissão, temporização e bizantina e se aplica a nodos ou mensagens no sistema. O modelo captura o comportamento usual de uma rede de comunicação onde mensagens podem ser perdidas (omissão), chegar atrasadas (temporização) ou ter seu conteúdo alterado (falha bizantina); nodos podem parar (colapso de nodo); links podem ser rompidos (colapso de link). Esse modelo simplificado permite construir injetores de falhas eficientes para o teste do comportamento sob falhas de sistemas distribuídos e para validar protocolos de comunicação. Exemplos de injetores que permitem emular falhas de acordo com o modelo são ORCHESTRA [Dawson 1996], NIST Net [Carson 2003], VirtualWire [De 2003], FIRMAMENT ([Drebes 2005][Siqueira, 2009]), FIONA [Jacques-Silva 2006] e FAIL-FCI [Horau 2007].

Atualmente, o projeto concentra-se em falhas de comunicação e injetores de falhas por software. A forma usual de injetar falhas de comunicação é interceptar mensagens enviadas e recebidas pela aplicação sob teste e, uma vez de posse de uma mensagem, decidir, de acordo com uma dada descrição de carga de falhas, se a mensagem vai ser descartada, atrasada ou corrompida. A aplicação sob teste pode ser um aplicativo de alto nível ou até a implementação de um protocolo de comunicação no nível do kernel. Para interceptar mensagens, são usados ganchos providos pelo sistema alvo como, por exemplo, bibliotecas de comunicação, chamadas de sistema, reflexão computacional, recursos de depuração ou filtros disponíveis no kernel do sistema.

Uma questão em aberto é se o modelo de falhas baseado em Cristian, e amplamente usado para redes nomádicas e sistemas distribuídos, é adequado a sistemas móveis. Vale notar que o modelo de Cristian não inclui, apesar de não excluir explicitamente, atenuação de sinal e variações crescentes e decrescentes no atraso ou perda de mensagens, o que pode corresponder a dispositivos afastando-se ou aproximando-se da base. Também não inclui particionamento de rede, o que é raro ocorrer em redes nomádicas, mas um evento comum em ambientes móveis [Oliveira 2009].

3 Plataformas móveis

Até pouco tempo atrás, toda a mobilidade ocorria apenas através de dispositivos como notebooks ou PDAs. Hoje, smartphones oferecem todos os recursos de um computador convencional [Oliver 2009] como sistema operacional, uma grande variedade de aplicativos, acesso a Internet, todos associados à comodidade de um telefone celular. Breve veremos crescer a diversidade de dispositivos móveis, como tablets e e-readers, de diversos fabricantes.

Várias plataformas móveis [Oliver 2009] são disponíveis para smartphones. A tabela 1 [Admob] mostra a participação no mercado mundial e no continente americano de cada um deles no último trimestre de 2009. Dados comparáveis para os primeiros três meses de 2010 ainda não estavam disponíveis quando o artigo foi escrito. Entretanto, durante o ano de 2009, a participação mundial do ambiente Android subiu de 1%, no primeiro trimestre, para 16%, no último trimestre. Permanecendo a tendência,

é possível já tenha ultrapassado o sistema Symbian da Nokia em participação no mercado mundial.

Outros tipos de dispositivos móveis também podem se beneficiar da disponibilidade dessas plataformas, principalmente se forem ambientes abertos.

Tabela 1: Participação no mercado por plataforma móvel para smartphones, Q4 2009

Sistema Operacional	Mercado Mundial (%)	América do Norte (%)	América Latina (%)
iPhone OS	51	54	56
Symbian	21	-	28
Android	16	27	1
RIM OS	6	10	8
Windows Mobile OS	3	3	6
Outros	3	6	3

Fonte: ADMOB, dezembro 2009

Apesar do mercado de aplicativos para plataformas móveis tender a um forte crescimento, com a consequente necessidade de teste sob falhas dessas aplicações, quase não foram encontradas ferramentas de injeção de falhas utilizadas exclusivamente em ambientes móveis. Uma exceção é a ferramenta mCrash [Ribeiro 2008]. Desenvolvida para o sistema operacional Windows Mobile 5, o mCrash permite testes automáticos para classes, métodos, parâmetros e objetos do framework .NET. Dinamicamente, mCrash gera scripts de teste, compila-os para o .NET, e invoca o processo de teste. Visando estritamente a validação de aplicações Java, parece possível uma adaptação de mCrash para o Android. No projeto, entretanto, pretende-se conduzir investigações com uma gama mais variada de aplicações, incluindo aplicativos de alto nível e recursos no nível de kernel.

3.1 Android

A Open Handset Alliance (OHA) é um grupo de mais de 40 empresas de áreas como operadores de telefonia, software e semicondutores. Liderada pela Google, a OHA visa criar padrões para a indústria da telefonia móvel. O primeiro passo nessa direção é o Android. Lançado oficialmente pela OHA em outubro de 2008, o Android é um ambiente de execução de aplicações para dispositivos móveis baseado no Linux [Chang 2010]. Aplicações podem ser desenvolvidas por terceiros, são facilmente integradas ao ambiente e têm acesso aos mesmos recursos que as aplicações originais dos fabricantes. Para facilitar a criação dessas aplicações, está disponível um kit de desenvolvimento de software (SDK). Entre os diversos componentes de desenvolvimento e depuração do SDK, pode-se destacar o emulador do Android e o conjunto de ferramentas para desenvolvimento integrado com o ambiente de desenvolvimento Eclipse [Eclipse].

O código fonte do Android [Android-source] é aberto e disponível sob licença GPL. O site de desenvolvimento para o Android [Dev-android] contém diversas informações para uso da plataforma de desenvolvimento [Android-git], incluindo as bibliotecas específicas para o sistema, o compilador adequado e os headers do sistema operacional.

3.2 Arquitetura do Android

A arquitetura do ambiente Android é mostrada na figura 1 [Chang 2010]. As suas camadas são:

- Aplicações escritas em Java (como cliente de e-mail, programa de SMS, calendário, mapas, contatos, entre outros).
- Framework de aplicação. Entre os serviços disponibilizados estão: provedores de conteúdo, que compartilham os dados entre aplicativos; gerente de notificação, que disponibiliza diferentes tipos de alertas para as aplicações; e gerente de atividades, que gerencia do ciclo de vida das aplicações.
- Bibliotecas básicas acessíveis através do framework de aplicação, como uma versão compacta da *libc* para sistemas embarcados, bibliotecas de mídia, gerente de navegação (acesso ao display), SQLite (base de dados relacional).
- Android *runtime*. Cada aplicação do Android roda o seu próprio processo, com uma instância própria na máquina virtual Dalvik. A Dalvik requer pouca memória e permite que múltiplas instâncias de sua máquina virtual executem ao mesmo tempo. As aplicações são compiladas em Java e traduzidas para o formato .dex por uma ferramenta do SDK.
- Linux Kernel, baseado no kernel versão 2.6. Usa os serviços de gerência de memória e processos, pilha de rede e controladores de dispositivos. Além disso, o kernel atua como uma camada de abstração entre o hardware e o resto do sistema.

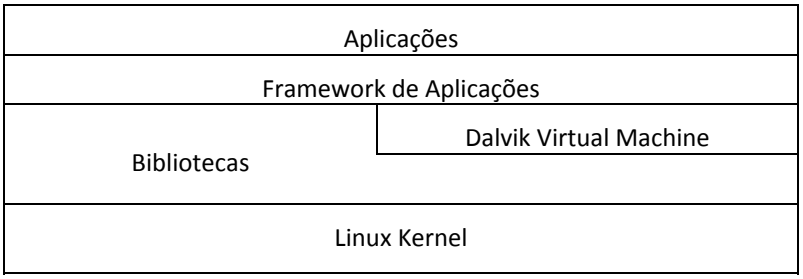


Figura 1: Arquitetura do ambiente Android

Um detalhe a ser ressaltado é que o ambiente Android não oferece a biblioteca *libc* completa, somente uma versão compacta da mesma, chamada Bionic. Esse problema é o maior empecilho para o reaproveitamento de códigos desenvolvidos para outros sistemas Linux. As razões para utilizar uma nova biblioteca em detrimento a *libc* são devidas principalmente: ao uso da licença BSD (se fosse empregada a *libc* original, a licença GPL contaminaria o espaço de usuário); à necessidade de uma biblioteca com tamanho reduzido que deve ser carregada em cada processo (a Bionic usa somente em torno de 200KB); e à inclusão de funções específicas para a melhor o desempenho do Android.

3.3 Sistema de desenvolvimento e emulação para o Android

O Kit de Desenvolvimento de Software (SDK) foi lançado junto com o Android. A versão 1.6, usada no projeto, é composta por: Android Virtual Devices (AVDs), Emulador, o Android Debug Bridge (ADB), entre outras.

Um dos principais componente do SDK é o emulador. Todo o desenvolvimento de aplicativos para o Android pode ser feito para rodar nesse emulador, que executa em computadores convencionais. O emulador do Android executa uma máquina virtual chamada Goldfish que opera sobre instruções ARM926T e que disponibiliza ganchos para entrada e saída de dados. Nela há arquivos específicos para a exibição na tela, o teclado do dispositivo é emulado pelo teclado do computador host, e a tela *touch screen* é emulado pelo mouse. Essas interfaces são usadas somente pelo emulador, não sendo compiladas quando o código deverá rodar em um dispositivo físico. Note-se que se pode fazer o download dos códigos fontes do kernel, para o emulador ou para os dispositivos, no mesmo link da plataforma de desenvolvimento. A versão usada para este trabalho foi a 2.6.29.

O emulador provê os componentes típicos de um aparelho celular, como botões de navegação, um teclado, e até mesmo uma tela que pode ser clicada com o mouse. No entanto, há algumas limitações, como não ter suporte a conexões USB, à câmera, a fones de ouvido, ao estado de carga da bateria, a Bluetooth e, é claro, a realização de chamadas telefônicas. As chamadas podem ser simuladas através da geração de um evento específico no console do emulador. O emulador funciona como um processo normal da máquina de desenvolvimento e comunica-se através de um roteador virtual, o qual não tem acesso direto à rede. Assim sendo, ele está sujeito às mesmas limitações que a sua rede impõe aos outros processos. O roteador suporta todos os tráfegos TCP e UDP do emulador. Contudo, outros protocolos, como o ICMP, ainda não são suportados. Atualmente, a falta de suporte do emulador se estende para mensagens multicast e IGMP.

3.4 ADB - Android Debug Bridge

O Android Debug Bridge permite a manipulação de um emulador ou um aparelho físico que esteja conectado à máquina de desenvolvimento. Ele é um programa cliente/-servidor baseado em três componentes: cliente, servidor e Daemon. O cliente e o servidor são executados na máquina de desenvolvimento. O servidor é responsável pela intermediação dos comandos originados do cliente e enviados ao Daemon. O Daemon é executado no emulador ou no aparelho móvel como um processo em background.

Ao se inicializar um cliente ADB, será feita a busca por um servidor operacional. Não encontrando um, o cliente vai disparar uma instância de servidor. Quando rodando, o servidor utiliza a porta TCP 5037 da máquina de desenvolvimento para receber comandos dos clientes. Todos os clientes enviam suas mensagens para essa mesma porta. Ao inicializar-se um servidor, são varridas todas as portas pares entre 5554 e 5584, buscando por um emulador ou dispositivo físico conectado. Naquela porta onde for encontrado um Daemon ADB, o servidor irá conectar-se. Observe que, quando for executado um emulador ou conectado um dispositivo físico à máquina de desenvolvimento, serão ocupadas duas portas em sequência: a porta par para o emulador ou dispositivo e a porta ímpar para o seu ADB.

4 Injetor de falhas para o Android

Para efetuar as primeiras avaliações da viabilidade de desenvolver um injetor de falhas em um dispositivo móvel, decidiu-se adaptar um injetor já existente ao novo ambiente. Dessa forma, as dificuldades do trabalho seriam restritas àquelas oferecidas pelo ambiente móvel e não pelo injetor. Adicionalmente, seria possível identificar quais

decisões de projeto usadas para implementar o injetor poderiam ser inconvenientes e, então, possibilitar uma especificação mais adequada de um novo injetor.

Entre os injetores disponíveis no grupo de pesquisa, um deles opera no kernel e vários outros operam com recursos de interceptação em Java, usando reflexão computacional ou recursos de depuração da máquina virtual. Esses últimos injetores são específicos para um determinado grupo de protocolos (TCP, UDP ou RMI isoladamente ou com os três protocolos simultaneamente), podem ser estendidos para novos protocolos de mais alto nível, mas visam exclusivamente o teste de aplicações escritas em Java. Como foram todos desenvolvidos também em Java, eles não são muito úteis para avaliar as peculiaridades das camadas de mais baixo nível do ambiente Android. O injetor ideal, neste caso, é o injetor que opera no kernel.

4.1 Injetor base: FIRMAMENT

A comunicação na Internet utiliza o modelo TCP/IP para troca de mensagens. Em consequência disso, as alterações feitas em pacotes IP irão se refletir em todos os outros protocolos encapsulados nele. Devido a esse uso tão comum da pilha TCP/IP, o protocolo IP está implementado no kernel do Linux e, dessa forma, atuando a partir do nível IP no kernel é possível injetar falhas em qualquer protocolo que rode sobre o IP.

Infelizmente, atuando a partir do kernel perdem-se muitas das vantagens associadas à injeção de falhas nos níveis de abstração mais altos [Menegotto 2007]. Para poder aproveitar tais vantagens, o porte e adaptação desses injetores de nível mais alto estão previstos para ser efetuados posteriormente no projeto, mas apenas quando o modelo de falhas para ambientes móveis tiver sido estabelecido e as adaptações necessárias realizadas.

O injetor FIRMAMENT [Drebes 2006] opera como um módulo no kernel e foi desenvolvido para a avaliação de protocolos de comunicação com o mínimo de interferência possível nas aplicações sob teste e para o máximo de abrangência em relação aos protocolos nos quais injetar falhas. Para tanto, é usada uma interface de programação do kernel, o Netfilter [Russel 2002], que fornece ganchos para a pilha de protocolos TCP/IP. Somente alguns dos ganchos disponíveis no Netfilter são necessários pelo injetor. De forma semelhante, não foram necessárias todas as funções disponibilizadas pelo Netfilter, tendo sido usadas apenas aquelas necessárias para realizar injeção de falhas.

Para descrever cargas de falha, o FIRMAMENT introduziu o conceito do *faultlet*. Um *faultlet* pode especificar atraso, duplicação ou descarte de um pacote, entre outras ações possíveis, sobre um dado pacote selecionado. O *faultlet* pode também especificar condições de seleção dos pacotes. Essas características permitem a criação de estruturas complexas para a injeção de falhas. Os *faultlets* podem ser configurados de forma independente para os fluxos de entrada e saída do protocolo IP. A máquina virtual FIRMVM tem a função de executar o *faultlet*, o qual é associado ao pacote e às variáveis de estado. Ela trabalha sobre as entradas e saídas de mensagens dos protocolos IPv4 e IPv6, sendo esses os pontos de injeção de falhas. A cada fluxo está associado um conjunto de variáveis de estado que é formado por 16 registradores de 32 bits. Essa máquina virtual é a que foi portada para o ambiente Android.

5 Sistema de desenvolvimento

Para que fosse possível instalar o injetor de falhas no Android, foram usadas as ferramentas descritas anteriormente. Além disso, foram seguidos os procedimentos documentados para a correta instalação dos módulos do injetor no ambiente Android. Um ponto a favor do Android é a disponibilidade de informação para desenvolvimento no ambiente. As dificuldades iniciais decorrentes do desconhecimento de um novo sistema foram sanadas com o estudo dos manuais de desenvolvimento. Quando esses não eram suficientes, as listas de discussão [Android-kernel] possibilitaram superar as dificuldades.

Uma das dificuldades enfrentadas foi o aprendizado da compilação cruzada (*cross compilation*) e das diferenças entre as formas de geração de código para o emulador ou para um dispositivo físico.

5.1 Porte do injetor através do sistema de desenvolvimento

O ambiente de desenvolvimento utilizado foi um computador com processador Intel E7400, sistema operacional Windows (host) e uma máquina virtual VirtualBox v2.2.2 executando Ubunto versão 8.04 e kernel 2.6.24. Dessa forma, o desenvolvimento é feito em um desktop, os fontes devem ser compilados no desktop tendo como alvo o Android, e o código-objeto deve ser adequadamente transferido para o Android (ou para o simulador). Os procedimentos, em termos gerais, são os seguintes:

- Obter os fontes da plataforma de desenvolvimento, do kernel (Goldfish) e o SDK do Android;
- Cross-compilar os fontes do FIRMAMENT, para que possam executar no Android;
- Recompilar o kernel do Android. Esse procedimento é necessário, pois o kernel do Android não tem, por padrão, o Netfilter, que é essencial para a operação do injetor;
- Criar um AVD (Android Virtual Device), para possibilitar executar o kernel e o injetor em um emulador de um dispositivo real, capaz de rodar o Android.

Uma vez obtidos todos os arquivos necessários para operação do injetor no emulador do Android, deve-se seguir os seguintes procedimentos:

- Iniciar, no desktop, uma instância do emulador, indicando o AVD criado e o kernel compilado;
- Iniciar, em outro terminal, o ADB (Android Debug Bridge), de maneira a poder interoperar com o emulador (ou o equipamento que roda o Android);
- Efetuar a transferência dos arquivos desejados para o emulador (ou equipamento) usando o comando *push* do ADB.

O detalhamento desses procedimentos pode ser encontrado no mesmo site onde estão o código fonte e as ferramentas de desenvolvimento [Dev-android].

5.2 Aplicações Java versus Aplicações “C”

O Android é baseado no kernel 2.6 do Linux. Ele substitui alguns módulos (a *libc*, por exemplo), mas o comportamento e a operação geral são semelhantes. Essa informação não é importante para quem desenvolve em Java (usando o Eclipse, por exemplo).

Entretanto, ela abre uma porta para aqueles que desenvolvem em ambiente Linux, pois, idealmente, deveria ser possível portar as aplicações que rodam em um Linux com kernel 2.6 para o Android. Infelizmente, como já foi dito, nem sempre isso é possível. E um dos motivos é o uso da biblioteca Bionic em lugar da *libc*. De qualquer forma, mesmo com essa limitação, o porte das aplicações escritas em "C" para o Android é menos oneroso do que um novo desenvolvimento, mesmo que sejam necessários alguns ajustes. No caso do FIRMAMENT, cujo código está escrito em "C" e roda sob o kernel 2.6, o porte aconteceu sem a necessidade de ajustes significativos.

5.3 Compilando o Kernel e transferindo arquivos para o Android

Nas primeiras tentativas de compilação do kernel, seguiu-se a documentação [Motz]. Entretanto, devido ao fato dessa documentação estar desatualizada, os compiladores e bibliotecas não são os mais indicados. Atualmente, é recomendada a utilização do kernel, compilador e bibliotecas específicas do Android [Android-git].

Após compilar o kernel e o FIRMAMENT, os arquivos objeto foram transferidos para seus devidos lugares, no sistema de arquivos do Android, e o emulador foi iniciado, tendo suas funções básicas operado corretamente.

6 Experimentos com o injetor de falhas no Android

São apresentados os experimentos realizados para comprovar se o porte foi efetivo e se o comportamento do injetor não sofreu alterações quando executado no novo ambiente. Esse foi o primeiro passo visando introduzir, no futuro, as adaptações necessárias no novo injetor, de maneira a incorporar um modelo de falhas adequado a ambientes móveis. O ambiente de testes é um computador com processador Intel E7400, sistema operacional Windows como hospedeiro de uma máquina virtual VirtualBox V2.2.2 executando Ubuntu versão 8.04 kernel 2.6.24.

O primeiro experimento injetava falhas em pacotes TCP e foi realizado para verificar se o FIRMAMENT e o Netfilter estavam minimamente operacionais. Os dois últimos experimentos foram realizados injetando-se falhas em mensagens UDP. Em todos os experimentos, o servidor foi executado no emulador e o cliente na máquina hospedeira do desenvolvimento. Como o servidor envia mensagens diretamente para um IP e porta específicos, caso se desejasse inverter a situação, com o cliente no emulador, seria necessário fazer o redirecionamento de portas da máquina de desenvolvimento para a porta em que o cliente estaria sendo executado no Android.

Convém observar que nem todas as funcionalidades do injetor foram convenientemente testadas no novo ambiente. É possível que algumas surpresas apareçam no decorrer de novos testes, mas os experimentos que apresentamos neste artigo são uma amostra significativa da compatibilidade do Linux do Android com os sistemas convencionais.

6.1 Operação básica do injetor no ambiente Android

Visando verificar se o FIRMAMENT e o Netfilter estavam operacionais após serem portados para o ambiente Android, para o primeiro experimento construiu-se um *faultlet* que selecionava todas as mensagens TCP e injetava falhas de descarte em todas as mensagens selecionadas. O resultado foi que o emulador travou totalmente, não sendo mais possível qualquer tipo de interação entre o ADB e o emulador.

Depois de alguma investigação, identificou-se que toda a comunicação entre ADB e emulador é feita através de mensagens TCP, apesar do ADB e o emulador estarem rodando na mesma máquina física. O mesmo ocorreria caso fosse tentada a comunicação com um dispositivo físico, Com isso pode-se identificar que toda a comunicação entre o ADB (rodando em uma máquina de desenvolvimento) e o Android (rodando como emulador ou em um dispositivo físico) utiliza uma única pilha TCP, de maneira a reduzir ao máximo o uso dos recursos dos dispositivos. Portanto, no nível do kernel, não existe discriminação entre mensagens de aplicação (navegador, por exemplo) e de sistema, como são as mensagens de controle e depuração trocadas com o ADB.

Portanto, numa segunda rodada deste experimento, para verificar se o injetor estava operando corretamente, o *faultlet* inicial foi alterado de maneira a descartar as mensagens dirigidas a todos os endereço IP, exceto aquele onde estava rodando o ADB. O resultado foi que a comunicação entre o emulador do Android e o ADB permaneceu operacional, mas não era possível navegar na internet usando o navegador do Android, como esperado. O experimento mostrou que o porte foi realizado com sucesso, mas os experimentos devem ser conduzidos com cuidado e os *faultlets* devem levar em consideração o ambiente de emulação e suas particularidades.

6.2 Descarte de Pacotes

Esse experimento mostra a capacidade do injetor em descartar pacotes UDP, quando operando no emulador. O injetor executa um *faultlet* que primeiro identifica o tipo de pacote que se deseja capturar, no caso, um pacote UDP. Em seguida é verificado o endereço de destino do pacote, pois foi decidido carregar o *faultlet* no fluxo de saída *ipv4_out*. Caso fosse colocado no fluxo de entrada, deveria se avaliar o remetente do pacote. Por último, para emular o descarte estatístico de pacotes, é feita a escolha dos pacotes a serem descartados com um fator aleatório e uniforme de 10% dos pacotes enviados.

A comunicação foi estabelecida através de um sistema cliente/servidor. O servidor foi acionado no emulador Android e enviou as mensagens para o cliente que estava na máquina hospedeira Windows. Para monitoração das mensagens perdidas, todas as mensagens enviadas continham uma numeração sequencial, lida pelo cliente. Por sua vez, o cliente também fez uma contagem de mensagens recebidas. O servidor foi programado para o envio de 10000 mensagens, com um tempo de 100ms entre cada uma, o que correspondeu à carga de trabalho do experimento.

A tabela 2 mostra o resultado médio das rodadas com e sem a atuação do injetor FIRMAMENT portado para o Android. Observa-se que a quantidade média de pacotes descartados foi um pouco superior aos 10% previstos, mas isso se deve principalmente ao procedimento de randomização usado para alcançar uma distribuição uniforme de probabilidade de perda de mensagem descrita no *faultlet*.

Tabela 2: Resultado do descarte dos pacotes

Rodada	Pacotes recebidos (%)	Tempo para envio (s)
Sem ação do injetor	100	1020
Com ação do injetor	89,63	1022

As rodadas foram executadas um número significativo de vezes de maneira a obterem-se valores minimamente confiáveis para verificar se o injetor operando no emulador Android conseguia efetivamente descartar mensagens, o que foi comprovado pelo experimento. Não era objetivo do experimento a obtenção de medidas precisas para uma avaliação de desempenho.

6.3 Atraso de Pacotes

Esse experimento simula o atraso de pacotes enviados pela rede. É muito comum a ocorrência de atrasos nas transmissões de dados, seja por causa de tráfego, ou mesmo devido ao atraso no processamento do pacote recebido. O experimento injeta um atraso variável de 12 ±5ms em cada mensagem selecionada. O valor do atraso foi escolhido unicamente para facilitar as medidas, não correspondendo a um cenário de falhas representativo de ambientes móveis. No estágio atual do projeto esse cenário de falhas representativo ainda não é conhecido.

No experimento, o envio e recebimento dos pacotes foram feitos com um sistema do tipo cliente/servidor. O servidor foi executado no emulador, enquanto o cliente foi executado na máquina hospedeira Windows. O servidor envia uma mensagem para o cliente e aguarda a resposta. O cliente recebe uma mensagem e envia uma resposta ao servidor. No total, o servidor envia 10000 mensagens, o que corresponde à carga de trabalho do experimento.

No *faultlet* usado para descrever a carga de falhas foi especificado um filtro do tipo de pacote que sofrerá a ação do injetor, no caso pacotes UDP. Em seguida selecionam-se somente as mensagens destinadas ao IP 155.19.72.189. Caso ambas as condições sejam verdadeiras, então o pacote é atrasado. Após ser confirmado o tipo de pacote e o endereço do destinatário, é feito um sorteio com números de -5 a +5. O resultado é somado ao valor constante 12 que fica armazenado em um registrador da máquina virtual do injetor. O valor desse registrador será usado como o atraso para o pacote em milissegundos.

A tabela 3 mostra os tempos de execução de rodadas sem e com a ação do injetor. Neste último têm-se o tempo de execução inserindo-se um atraso variável de 12±5ms. Os resultados permitem concluir que o injetor portado para o Android estava efetivamente promovendo o atraso variável no envio de mensagens.

Tabela 3: Resultado do atraso das mensagens

Rodada	Tempo para envio (s) de 10000 mensagens
Sem ação do injetor	1024
Com ação do injetor (atraso variável)	1127

Assim como no primeiro experimento, as rodadas foram executadas um número de vezes adequado apenas para se obter valores minimamente confiáveis. Não era objetivo do experimento a obtenção de medidas precisas de tempo, apenas verificar que o injetor estava efetivamente injetando atrasos variáveis nas mensagens, como seria esperado se o porte tivesse sido bem sucedido.

7 Trabalhos Futuros

Na continuidade do trabalho desenvolvido pretende-se atuar em duas áreas. A primeira é na direção do estudo do comportamento sob falhas das aplicações que visam fornecer dependabilidade para os dispositivos móveis e a segunda é definir modelos de falha de mobilidade, que serão usados para projetar um novo sistema de injeção de falhas.

O estudo experimental da dependabilidade provida pelas aplicações será feito através de uma série de experimentos de injeção de falhas e monitoração de comportamento, formando um Benchmark de Dependabilidade. Com isso, os usuários e desenvolvedores poderão avaliar a dependabilidade das aplicações, identificar problemas de implementação e comparar diferentes soluções para o mesmo problema, possibilitando que o produto final seja de melhor qualidade. Espera-se que, como resultado desse desenvolvimento, obtenha-se um benchmark que possa ser utilizado pelos desenvolvedores para alcançar aplicações mais robustas, mesmo sem conhecer os aspectos específicos do comportamento das falhas.

No outro ramo de desenvolvimento, o objetivo é modelar de forma mais precisa as falhas que ocorrem nos sistemas móveis. Os modelos resultantes desses estudos deverão ser incorporados aos injetores de falhas de maneira a possibilitar uma reprodução mais fiel dos cenários de falha reais. Acredita-se que os modelos de falha dos sistemas móveis sejam diferentes daqueles encontrados nos sistemas nomádicos. Entretanto, não se pode descartar a possibilidade de emular as falhas típicas dos sistemas móveis através de falhas encontradas nos sistemas nomádicos. A verificação dessa possibilidade assim como a necessidade de construção de um novo injetor de falhas é o resultado esperado nessa linha de investigação.

8 Conclusão

O artigo apresentou os primeiros estudos de uma nova linha que estamos investigando que inclui a validação de aplicações em ambientes móveis sujeitas a falhas de comunicação. Apresentou também os resultados da primeira tarefa deste novo projeto que foi o porte de um injetor de falhas de comunicação, que opera como um módulo no nível do kernel Linux, para o ambiente Android.

A experiência com o novo ambiente foi considerada muito satisfatória. As dificuldades encontradas foram mais devido a pouca familiaridade com o ambiente e imprecisões na documentação do que a restrições ou bugs de implementação do Android. Não é objetivo do projeto testar o ambiente Android, que julgamos ser suficientemente robusto e confiável. O objetivo é avaliar aplicações desenvolvidas para o ambiente Android. Uma grande quantidade dessas aplicações tem sido disponibilizada, tais aplicações surgem vindas das mais diversas fontes. Um benchmark que possa ajudar a avaliar a dependabilidade de aplicações para ambientes móveis é útil não apenas a desenvolvedores, mas também a usuários ou sistemas que precisam depositar confiança no funcionamento correto dessas aplicações.

9 Referências bibliográficas

Admob (2010). admob-mobile-metrics-report-dezember-09. Disponível em: < http://metrics.admob.com/ >. Acessado em: março, 2010.

Android-ADB (2009). Disponível em: < http://developer.android.com/guide/developing/tools/adb.html >. Acessado em: setembro, 2009

Android-git (2009). Disponível em: < http://android.git.kernel.org/ >. Acessado em: setembro, 2009.

Android-kernel (2009). Disponível em: < http://groups.google.com/group/android-kernel >. Acessado em: setembro, 2009.

Android-source (2009). Disponível em: < http://source.android.com/ >. Acessado em: junho, 2009

Arlat, J.; Aguera, M.; Amat, L.; Crouzet, Y.; Laprie, J.; Fabre, J.; Martins, E.; Powell, D. (1990). Fault-injection for dependability validation: a methodology and some applications. IEEE Trans. on Soft. Eng., Special Issue on Experimental Computer Science, 3, vol. 16, n.2, p. 166-82, Feb. 1990.

Avizienis, A.; Laprie, J.; Randell B,; Landwehr C. (2004). Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE trans. on dependable and secure computing, vol. 1, n. 1, jan 2004, pp 11-33

Carson, M.; Santay, D. (2003). NIST Net – A Linux-based Network Emulation Tool. ACM SIGCOMM Computer Communications Review, vol.33, pp.111-126, 2003.

Chang, G.; Tan. C.; Li, G.; Zhu, C. (2010). Developing Mobile Applications on the Android Platform. In: Mobile Multimedia Processing, LNCS 5960, Springer, pp. 264–286

Cristian, F. (1991). Understanding fault-tolerant distributed systems. Communications of the ACM, vol. 34, n.2, p. 56-78

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

De, P.; Anindya Neogi, Tzi-cker Chiueh. (2003). VirtualWire: A Fault Injection and Analysis Tool for Network Protocols. 23rd IEEE International Conference on Distributed Computing Systems (ICDCS'03), pp.214

Dev-android (2009). Disponível em: < http://developer.android.com/index.html >. Acessado em: agosto, 2009.

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

Eclipse (2009) Disponível em: < http://www.eclipse.org/>. Acessado em: agosto, 2009.

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

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

Jacques-Silva, G. ; Drebes, R. J. ; Gerchman, J. ; Trindade, J. ; Weber, T. S.; Jansch-Pôrto, I. (2006). A Network-level Distributed Fault Injector for Experimental Validation of Dependable Distributed Systems. In: 30th Annual International

Computer Software and Applications Conference – COMPSAC 2006, Chicago. IEEE Computer Society Press, 2006. v. 1. p. 421-428.

Krishna, P., Vaidya, N., Pradhan, D. (1993). Recovery in distributed mobile environments. In Proceedings of the IEEE Workshop on Advances in Parallel and Distributed Systems, pp. 83-88.

Menegotto, C. C.; Vacaro, J. C.; Weber, T. S. (2007). Injeção de Falhas de Comunicação em Grids com Características de Tolerância a Falhas. In: VIII Workshop de Teste e Tolerância a Falhas, 2007, Belém. WTF 2007 - VIII Workshop de Teste e Tolerância a Falhas. v. 1. p. 71-84.

Motz (2009). Disponível em: < http://honeypod.blogspot.com/2007/12/compile-android-kernel-from-source.html >. Acessado em: agosto, 2009.

Oliveira, G. M.; Cechin, S.; Weber, T. S. (2009). Injeção Distribuída de Falhas de Comunicação com Suporte a Controle e Coordenação de Experimentos. In: Workshop de Testes e Tolerância a Falhas, João Pessoa. WTF 2009, v. 1. p. 101-114

Oliver, E. 2009. A survey of platforms for mobile networks research. *SIGMOBILE Mob. Comput. Commun. Rev.* 12, 4 (Feb. 2009), 56-63.

Ribeiro, J.C. (2009). mCrash: a Framework for the Evaluation of Mobile Devices' Trustworthiness Properties; Organization: University of Coimbra, Portugal; Supervisor: Prof. Mário Zenha-Rela; Period: 2005-2008; Presentation Date: 17th of December, 2008.

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

Siqueira, T.; Fiss, B. C.; Weber, R.; Cechin, S.; Weber, T. S. (2009). Applying FIRMAMENT to test the SCTP communication protocol under network faults. In: 10th Latin American Test Workshop. v. 1. p. 1-6

Injeção de Falhas de Comunicação em Aplicações Java Multiprotocolo

Cristina Ciprandi Menegotto[1], **Taisy Silva Weber**[1]

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

{ccmenegotto,taisy}@inf.ufrgs.br

Abstract. *Some networked applications are based on more than one communication protocol, such as UDP, TCP and RMI, and must be carefully tested under communication faults. If the emulation of a fault that affects message exchanging does not take into account all simultaneously used protocols, the behavior emulated in an experiment can be different from that observed under real fault occurrence. This paper presents* Comform*, a communication fault injector for multi-protocol Java applications. It works at JVM level, intercepting protocol messages, and, in some cases, it also operates at the operating system level, using firewall rules. The approach is useful for both white box and black box testing and preserves the target application's source code.*

Resumo. *Algumas aplicações de rede são baseadas em mais de um protocolo de comunicação, como UDP, TCP e RMI e devem ser testadas cuidadosamente em presença de falhas de comunicação. Caso a emulação de uma falha que afete a troca de mensagens não considere todos os protocolos simultaneamente utilizados, o comportamento emulado poderá diferir do observado na ocorrência de uma falha real. Este artigo apresenta* Comform*, um injetor de falhas de comunicação para aplicações Java multiprotocolo que opera no nível da JVM, interceptando mensagens de protocolos, e, em alguns casos, opera também no nível do sistema operacional, usando regras de* firewall*. A abordagem é útil para testes de caixa branca e preta e preserva o código fonte da aplicação alvo.*

1. Introdução

Aplicações de rede com altos requisitos de dependabilidade devem ser testadas cuidadosamente em condições de falhas de comunicação para aumentar a confiança no seu comportamento apropriado na ocorrência de falhas. Injeção de falhas de comunicação é a técnica mais adequada para o teste dos mecanismos de tolerância a falhas destas aplicações. Ela é útil tanto para auxiliar na remoção de falhas como na previsão de falhas.

Dentre as aplicações de rede, algumas são baseadas em mais de um protocolo, como UDP, TCP e RMI. Elas são denominadas multiprotocolo no contexto desse trabalho, que foca naquelas escritas em Java e baseadas em protocolos que estão acima do nível de rede na arquitetura TCP/IP. Aplicações multiprotocolo são relativamente comuns, pois diferentes protocolos de comunicação podem ser empregados para diferentes propósitos em uma mesma aplicação de rede. Na literatura, alguns exemplos de aplicações Java multiprotocolo tolerantes a falhas são Zorilla [Drost et al. 2006], Anubis [Murray 2005] e algumas aplicações do *middleware* para comunicação de grupo JGroups [Ban 2002].

Tais exemplos fazem uso simultâneo dos protocolos, ou seja, podem usar mais de um em uma única execução.

Um injetor de falhas de comunicação adequado, que trate todos os protocolos utilizados, é necessário para o seu teste. Caso a emulação de uma falha que afete a troca de mensagens não leve em consideração todos os protocolos simultaneamente utilizados, o comportamento emulado durante um experimento poderá ser diferente daquele observado na ocorrência de uma falha real, de modo que podem ser obtidos resultados inconsistentes sobre o comportamento da aplicação alvo em presença da falha.

Muitos injetores de falhas de comunicação não são capazes de testar aplicações Java multiprotocolo. Outros possuem potencial para o teste dessas aplicações, mas impõem grandes dificuldades aos engenheiros de testes. Por exemplo, contrariamente ao enfoque deste trabalho, algumas ferramentas são voltadas à injeção de falhas de comunicação para o teste de protocolos de comunicação, e não de aplicações. Tal orientação ao teste de protocolos costuma levar a grandes dificuldades no *teste de caixa branca* de aplicações. Testes de caixa branca, também chamados de estruturais, levam em consideração o código fonte da aplicação alvo [Pezzé and Young 2008]. Entre outros exemplos de dificuldades proporcionadas por ferramentas da literatura estão a incapacidade de testar diretamente aplicações Java e a limitação quanto aos tipos de falhas que permitem emular. A análise de tais ferramentas motiva o desenvolvimento de uma solução voltada especificamente ao teste de aplicações multiprotocolo desenvolvidas em Java.

Este artigo apresenta uma solução para injeção de falhas de comunicação em aplicações Java multiprotocolo. A solução opera no nível da JVM, interceptando mensagens de protocolos, e, em alguns casos, opera também no nível do sistema operacional, usando regras de *firewall* para emulação de alguns tipos de falhas que não podem ser emulados somente no nível da JVM. Ela é útil para testes de caixa branca e preta e possui características importantes como a preservação do código fonte da aplicação alvo. A viabilidade da solução proposta é mostrada por meio do desenvolvimento de Comform (COMmunication Fault injector ORiented to Multi-protocol Java applications), um protótipo para injeção de falhas de comunicação em aplicações Java multiprotocolo que atualmente pode ser aplicado para testar aplicações Java baseadas em qualquer combinação dos protocolos UDP, TCP e RMI (incluindo as baseadas em único protocolo).

A seção 2 trata do problema da injeção de falhas de comunicação em aplicações Java multiprotocolo, define requisitos para uma solução, analisa o potencial de ferramentas da literatura para tratar deste problema e compara Comform aos trabalhos relacionados. A seção 3 apresenta um modelo genérico de solução, enquanto a seção 4 apresenta o injetor de falhas Comform. A seção 5 apresenta a condução de experimentos com Comform. Por fim, a seção 6 conclui o artigo.

2. Injeção de Falhas em Aplicações Java Multiprotocolo

Para avaliar o comportamento de uma aplicação Java multiprotocolo em presença de falhas de comunicação usando injeção de falhas, um injetor de falhas de comunicação adequado é necessário. É inconsistente testar uma aplicação em que há uso simultâneo de protocolos com um injetor de falhas voltado ao teste de aplicações Java baseadas só em UDP, como FIONA [Jacques-Silva et al. 2006], e depois testá-la com um injetor voltado ao teste de aplicações Java baseadas em TCP, como FIERCE [Gerchman and Weber 2006].

Para mostrar a inconsistência, Zorilla[Drost et al. 2006], um *middleware par-a-par*, implementado em Java, que visa à execução de aplicações de supercomputação em grades, pode ser considerado como alvo. Seu projeto é baseado em uma rede de nodos, todos capazes de tratar submissão, escalonamento e execução de *jobs* e armazenamento de arquivos. Um nodo faz *broadcast* de pacotes UDP periodicamente na rede local para procurar por outros nodos. O endereço de um ou mais nodos existentes da rede é necessário para se unir a ela. Usando TCP, nodos se conectam diretamente um a outro para envio de grandes quantidades de dados e também é permitido que programas se conectem a Zorilla para, por exemplo, submeter um *job*.

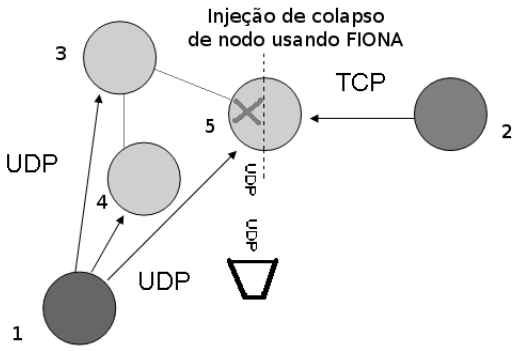


Figura 1. Inadequação de injeção de colapso em nodo Zorilla usando FIONA.

A Figura 1 exemplifica uma tentativa de injeção de colapso de nodo em Zorilla usando FIONA, que elimina a comunicação UDP entre o nodo onde o injetor é executado e os demais. Colapso de nodo refere-se à situação em que um computador pára sua execução e entra em colapso, sem reinicialização. Os nodos 3, 4 e 5 compõem uma rede Zorilla. O nodo 1 está fazendo *broadcast* de pacotes UDP com o objetivo de se unir à rede. O nodo 2 representa um programa externo conectando-se a Zorilla para submissão de um *job*. A injeção de um colapso no nodo 5 usando FIONA resultará em um teste inconsistente no qual o colapso não é percebido pelo nodo 2, pois só a comunicação UDP é suprimida. Problemas similares ocorrem usando um injetor voltado ao teste de aplicações baseadas só em TCP, como FIERCE.

2.1. Requisitos Identificados para a Solução

Além de ser capaz de injetar falhas em aplicações Java multiprotocolo, alguns outros requisitos importantes foram identificados para a solução. Tais requisitos, que ajudam a minimizar as dificuldades enfrentadas por um engenheiro de testes, são listados abaixo:

- Deve ser capaz de emular os tipos de falhas de comunicação que podem ocorrer comumente em ambientes de rede, como, por exemplo, colapsos de nodos, colapsos de *links*, falhas de temporização, omissão e particionamento de rede. Caso contrário, a cobertura dos testes pode ser pobre em muitos casos, como no teste de aplicações que possuem altos requisitos de dependabilidade.
- Deve preservar o código fonte da aplicação alvo, pois ele nem sempre está disponível e, mesmo se disponível, modificá-lo leva à grande intrusividade espacial.
- Deve ser capaz de injetar falhas tanto independentemente do conhecimento do código fonte da aplicação alvo (para testes de caixa preta) como também levando em consideração o código fonte da aplicação alvo (para testes de caixa branca).

- Deve prover um mecanismo adequado para descrição de cargas de falhas, evitando dificuldades que levem o engenheiro de testes à desistência da condução de certos experimentos.

A seguir, outras ferramentas de injeção de falhas são analisadas e é indicado como elas atendem ou não aos requisitos identificados para a solução.

2.2. Potencial de Injetores para o Teste de Aplicações Java Multiprotocolo

FIRMI [Vacaro 2007] é um injetor de falhas cujas aplicações alvo são as escritas em Java e baseadas em RMI, ou seja, ele também não é voltado ao teste de aplicações multiprotocolo. Embora FIRMI e as outras ferramentas citadas anteriormente não atinjam ao objetivo deste trabalho, elas preenchem alguns dos outros requisitos considerados importantes para uma solução e inspiraram a modelagem e desenvolvimento de Comform. Em especial, FIRMI é útil tanto para testes de caixa preta como de caixa branca.

As diferentes implementações do *framework* FIT [Looker et al. 2004] interceptam mensagens SOAP em sistemas baseados em *Web Services*, não tratando outros tipos de protocolo. Tais abordagens trabalham somente em um alto nível de abstração, não sendo adequadas aos propósitos deste trabalho.

NFTAPE [Stott et al. 2000] é um *framework* para avaliação de dependabilidade em sistemas distribuídos usando injeção de falhas. Diferente de outras abordagens, ele faz distinção entre *injetores leves* (*lightweight fault injectors* ou LFI) – componentes responsáveis pela injeção da falha – e *gatilhos* (*triggers*) – responsáveis por disparar a injeção de falhas. Para testar aplicações Java multiprotocolo com NFTAPE, seria necessário projetar e implementar corretamente estes componentes, já que não há relatos sobre sua existência na literatura. Em outras palavras, a abordagem de NFTAPE não provê uma solução para injeção de falhas de comunicação em aplicações multiprotocolo e um grande esforço seria necessário para criar componentes adequados.

FAIL-FCI [Hoarau et al. 2007] é um injetor de falhas para aplicações distribuídas. Ele pode ser usado para injeção de falhas em aplicações tanto de modo quantitativo, como qualitativo, e não requer a modificação do código fonte da aplicação alvo. FAIL-FCI tem potencial para o teste de aplicações Java multiprotocolo, mas com a forte desvantagem de que somente é capaz de emular colapsos de processos e suspensão de processos. Em geral, é muito importante testar aplicações de rede com mais tipos de falhas de comunicação, como colapsos de nodos, colapsos de *links* e falhas de omissão.

Loki [Chandra et al. 2004] é um injetor de falhas para sistemas distribuídos que leva em consideração o estado global do sistema para injeção de falhas. Ele tem potencial para o teste de aplicações multiprotocolo, mas leva a diversas dificuldades. Loki requer a modificação do código fonte da aplicação alvo. Ainda, foi implementado em C/C++ e não pode testar diretamente aplicações Java.

FIRMAMENT [Drebes 2005] é um injetor de falhas cujo propósito principal é testar protocolos baseados em IP. Ele é implementado como um módulo do núcleo Linux, de modo que somente sistemas que podem ser executados nesse ambiente podem ser testados com a ferramenta. FIRMAMENT intercepta o envio e recebimento de pacotes IP em um nodo. Cargas de falhas (*faultloads*), especificadas usando uma linguagem de *bytecode*, são capazes de alterar conteúdo de pacotes e retornar a ação final a ser realizada

sobre eles. FIRMAMENT pode ser usado no teste de aplicações Java multiprotocolo, mas somente teste de caixa preta é viável, pois é muito difícil construir *faultloads* nos quais falhas são ativadas em pontos específicos da execução de aplicações (já que o injetor é "cego" à semântica delas). Mesmo para teste de caixa preta de aplicações, pode ser muito difícil escrever *faultloads*, principalmente para aplicações baseadas em protocolos de mais alto nível, como RMI [Vacaro 2007]. Como todas as mensagens enviadas ou recebidas por um nodo são interceptadas, a seleção daquelas específicas de um determinado processo requer um grande esforço para codificação de faultloads. Ferramentas como FIONA e FIRMI, apesar de não serem multiprotocolo, visam ao teste de *aplicações* Java e superam muitas das dificuldades impostas por ferramentas que visam ao teste de *protocolos*.

2.3. Comparação de Comform a Trabalhos Relacionados

A Tabela 1 compara, sumariamente, Comform a Loki [Chandra et al. 2004], FAIL-FCI [Hoarau et al. 2007] e FIRMAMENT [Drebes 2005]. Contrariamente à abordagem de Comform, estas últimas três ferramentas não cumprem todos os requisitos identificados para a solução. Apesar disso, diferente das outras ferramentas apresentadas anteriormente, elas podem potencialmente testar aplicações multiprotocolo. Para as células sobre as quais não foi possível chegar a uma conclusão, foi atribuído o valor "?".

Tabela 1. Comparando Comform a trabalhos relacionados.

	Comform	FAIL-FCI	Loki	FIRMAMENT
Capaz de injetar falhas em aplicações Java	sim	sim	não	sim
Vários tipos de falhas de comunicação	sim	não	?	sim
Preserva código fonte inalterado	sim	sim	não	sim
Teste de caixa branca de aplicações	sim	sim	sim	não
Teste de caixa preta de aplicações	sim	sim	não	sim
Descrição facilitada de faultloads	sim	?	?	não

3. Modelo Genérico

Esta Seção apresenta um modelo genérico de solução para injeção de falhas de comunicação em aplicações multiprotocolo.

3.1. Cuidados para Emulação de Colapso em Aplicações TCP

Protocolos de transporte, como UDP e TCP, são implementados no núcleo do sistema operacional. A emulação de falhas de comunicação em aplicações Java baseadas *somente* em UDP pode ser feita completamente no nível da JVM [Jacques-Silva et al. 2006]. No caso de falhas de colapso de nodo, por exemplo, essa emulação pode ser feita por meio da inibição do envio e recebimento de mensagens UDP no nível da JVM. Isso é suficiente devido à simplicidade do protocolo UDP. Por outro lado, considerando o TCP e aqueles protocolos de mais alto nível que o têm como base, falhas de comunicação que envolvem o descarte de mensagens dificilmente podem ser emuladas somente no nível da JVM.

A Figura 2 explica, de modo simplificado, a inadequação de emular falhas de colapso de nodo (ou outros tipos de falhas caracterizados pelo descarte de mensagens) em aplicações Java baseadas em TCP, e/ou em protocolos baseados em TCP, apenas no nível da JVM. O nodo A está executando uma aplicação Java cliente, baseada em TCP, enquanto o nodo B está executando sua correspondente aplicação servidor. Uma conexão

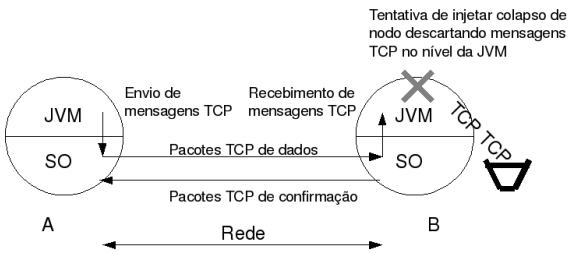


Figura 2. Emulação inconsistente de falhas de colapso no nível da JVM.

já foi estabelecida e tenta-se injetar uma falha de colapso em B por meio do descarte de mensagens TCP no nível da JVM de B. Pode-se observar o cliente enviando mensagens TCP ao servidor. Embora essas mensagens sejam descartadas no nível da JVM do servidor, o seu sistema operacional (SO) ainda irá enviar ao cliente confirmações relacionadas ao recebimento das mensagens, de modo que a falha não é emulada corretamente.

3.2. Seleção e Manipulação de Mensagens

Para que falhas que envolvem o descarte de pacotes sejam emuladas com representatividade em um nodo, pacotes relativos a todos os protocolos utilizados pela aplicação alvo devem ser passíveis de seleção e manipulação antes que sejam entregues à aplicação alvo (no caso de recebimento de pacotes) e antes que sejam enviados a outros nodos (no caso de envio de pacotes). Assim, essa seleção e manipulação deve ser feita no contexto do sistema operacional. Por outro lado, ao contrário de falhas caracterizadas pelo descarte de pacotes, falhas de temporização podem ser injetadas pela seleção e manipulação de mensagens em um nível mais alto de abstração.

Visando atender aos requisitos identificados para a solução, o modelo também requer a interceptação de mensagens em um nível de abstração mais alto do que o nível do sistema operacional de modo que informações específicas relacionadas à aplicação alvo possam ser facilmente recuperadas para a ativação e desativação de falhas. Por exemplo, considerando uma aplicação baseada em RMI, pode ser interessante a um engenheiro de testes ter a possibilidade de ativar uma falha antes da invocação de um determinado método remoto ou depois de um certo número de invocações de métodos remotos. Este tipo de informação é de difícil obtenção no nível do núcleo do sistema operacional, mas pode ser facilmente obtido em um nível mais alto de abstração.

4. Arquitetura de Comform

Esta Seção apresenta a arquitetura básica de Comform. Como descrito na Seção 3, o modelo genérico requer um modo de selecionar e manipular mensagens no nível do sistema operacional de forma a emular corretamente falhas que envolvem o descarte de pacotes (como falhas de colapso de nodos e *links*). A arquitetura usa um componente Firewall para esse fim. Esta abordagem foi aplicada com sucesso no injetor de falhas FIRMI e seu reuso mostrou-se conveniente. O uso de regras de *firewall* para o descarte de pacotes evita problemas como o descrito na Figura 2, pois os pacotes são descartados pelo próprio sistema operacional, antes que eles sejam entregues à aplicação e antes que sejam enviados a outros nodos. Deste modo, os nodos que devem perceber o estado falho do nodo onde a injeção está sendo realizada o farão de modo consistente.

O modelo também requer um modo de interceptar mensagens em um nível de abstração mais alto que o do sistema operacional, tal que informações específicas relacionadas à aplicação alvo possam ser facilmente recuperadas para *ativação* e *desativação* de falhas em testes de caixa branca. Seguindo a abordagem usada em outras ferramentas ([Jacques-Silva et al. 2006], [Vacaro 2007]), classes que implementam os protocolos de interesse – UDP, TCP e RMI – são instrumentadas no nível da JVM. Com a sua instrumentação, é possível obter informações de interesse para ativação e também para emulação de falhas. Para promover interação entre as partes da arquitetura que operam nos níveis da JVM e do sistema operacional, informações de portas locais sendo utilizadas pela aplicação alvo são obtidas por meio da instrumentação e usadas para a construção de regras de *firewall*. Ainda, falhas de temporização podem ser emuladas no nível da JVM.

Os tipos de falhas de comunicação que podem, atualmente, ser emulados por Comform incluem colapso de nodos, colapso de *links* e falhas de temporização. O protótipo pode ser estendido com a inclusão de novos tipos de falhas, como falhas de omissão, mas já possui uma variedade de tipos de falhas mais rica do que a oferecida por ferramentas como FAIL-FCI [Hoarau et al. 2007].

A Figura 3 apresenta uma visão simplificada da arquitetura de Comform. Uma linha tracejada separa o nível da JVM do nível do sistema operacional (SO). O *Firewall* e as implementações dos protocolos TCP e UDP na pilha de protocolos TCP/IP operam no nível do SO. No nível da JVM, a Aplicação Alvo é executada e as classes de comunicação de interesse da API Java são instrumentadas em tempo de carga. A instrumentação dessas classes provê a interação com o Controlador do injetor de falhas. Este componente é responsável pelo controle dos experimentos e interage com os outros componentes do injetor. O Monitor coleta informações importantes sobre o experimento. A Carga de Falhas (*Faultload*) inclui as noções de Carga de Falhas propriamente dita e de Carregador de Carga de Falhas. Em Comform, *Faultloads* são descritos como classes Java e um Carregador de Carga de Falhas é responsável por sua carga. Esses conceitos são reusados de FIRMI (e também foram reusados em um trabalho relacionado [Cézane et al. 2009]). A caixa Falha representa as falhas que a ferramenta é capaz de emular. Por fim, o Filtro de Mensagens é responsável pela efetiva injeção das falhas especificadas por um módulo de Carga de Falhas. A interface *Firewall* representa a interação com um *Firewall* de modo a emular corretamente falhas caracterizadas pelo descarte de mensagens.

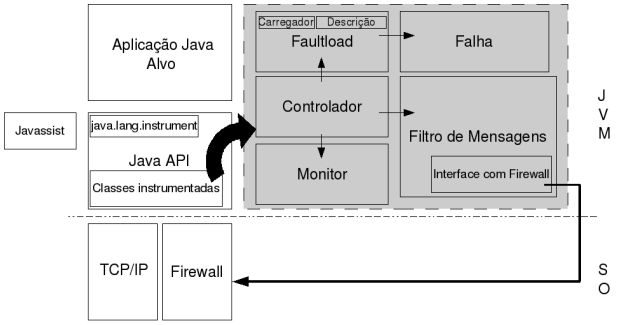


Figura 3. Arquitetura simplificada da ferramenta.

O pacote `java.lang.instrument` [SUN MICROSYSTEMS 2008] é utilizado na arquitetura para a interceptação do carregamento de classes na JVM. Deste modo,

é obtido acesso ao *bytecode* de classes Java que implementam os protocolos de interesse. A adição de código especial nessas classes, ou seja, a instrumentação dessas classes, é necessária para que elas interajam com o Controlador do injetor de falhas quando seus objetos forem invocados pela aplicação alvo. Apesar de prover acesso às classes de interesse e possibilitar adição de *bytecodes*, o pacote não é capaz de realizar a instrumentação de código propriamente dita. Deste modo, uma biblioteca especializada em instrumentação de *bytecodes* deve ser selecionada para essa tarefa. Javassist [Chiba 1998] foi escolhida para esse propósito. Como ela não faz parte da API de Java, aparece representada na Figura 3 como uma caixa separada.

O funcionamento básico do injetor pode ser resumido da seguinte forma:

1. Uma instância do Controlador (classe `Controller`) é obtida e é feita instrumentação das classes de interesse dos protocolos alvo durante o seu carregamento. No contexto da criação de uma instância de `Controller`, é feito o carregamento do *Faultload* a ser utilizado no experimento (classe derivada da classe `Faultload` desenvolvida pelo usuário) e execução de seu método construtor. O carregamento do *Faultload* é feito por um Carregador de Módulos de Carga de Falhas (classe `BaseFaultloadLoader`).
2. Quando é realizada a associação de um `socket` a uma porta local no nodo o injetor está sendo executado, a porta local é registrada no Filtro de Mensagens (classe `MessageFilter`).
3. Quando uma mensagem dos protocolos de interesse é interceptada, ela é processada pela classe `Controller`, que aciona a coleta de dados pelo Monitor (classe `Monitor`), registra a mensagem em *log*, invoca o método `update` do `Faultload` para que sejam realizadas possíveis decisões sobre ativação de falhas (podendo considerar o conteúdo da mensagem) e, finalmente, injeta as possíveis falhas ativadas no passo anterior por meio da classe `MessageFilter`.

Falhas podem ser ativadas durante a carga do injetor de falhas, ficando ativas desde o início do experimento, ou mais adiante, durante a execução da aplicação alvo. Neste caso, a ativação pode levar em consideração o conteúdo e a quantidade de mensagens interceptadas. Os atributos de mensagens dos protocolos de interesse podem ser usados para propósitos de ativação de falhas. A Figura 4 apresenta a modelagem das mensagens dos protocolos de interesse. Mensagens UDP e TCP possuem como atributos o tipo de mensagem, os dados sendo enviados ou recebidos, o tamanho da mensagem, o endereço de rede do nodo remoto e a porta do nodo remoto. Uma requisição RMI é representada pela classe `RMIRequest` e possui como atributos principais o tipo de requisição, o endereço de rede do nodo remoto, a referência remota usada, o método que está sendo invocado e os valores dos parâmetros deste método. É possível construir um Módulo de Carga de Falhas que ative falhas com base, por exemplo, no nome de um método sendo invocado.

Os atributos correspondentes a tipos de mensagem UDP e TCP ou tipo de requisição RMI podem assumir vários valores. A Tabela 2 mostra esses valores para TCP, considerando o pacote `java.nio`, e para RMI (os valores para TCP no pacote `java.net` e para UDP em ambos os pacotes seguem lógica semelhante). Para cada método relacionado a envio ou recebimento de mensagens (ou estabelecimento de conexão), existe a possibilidade de realizar ativação de falhas antes de sua execução (ao início da execução do método) e após sua execução (logo antes do método encerrar

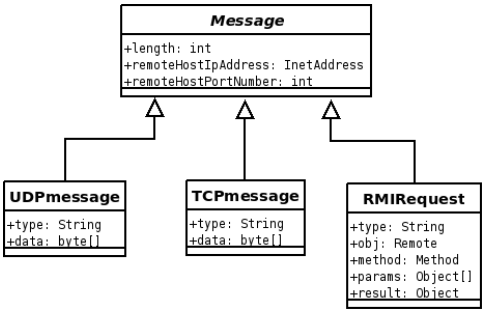


Figura 4. Modelagem de mensagens dos protocolos de interesse.

sua execução e retornar). Por exemplo, é possível realizar ativação de falhas antes da execução de um método remoto por um servidor ("RMI_beforeExecuting") ou logo antes da execução do método remoto retornar ("RMI_afterExecuting"). Também é possível ativar falhas com base em informações coletadas pelo Monitor, como o total de *bytes* enviados ou recebidos ou o total de requisições RMI efetuadas.

Tabela 2. Valores para o atributo tipo (`type`) de mensagens TCP (java.nio) e RMI.

Métodos TCP instrumentados (java.nio)	**Valores para atributo tipo**
connect	"TCPNio_connect_before", "TCPNio_connect_after"
long write	"TCPNio_writeLong_before", "TCPNio_writeLong_after"
int write	"TCPNio_writeInt_before", "TCPNio_writeInt_after"
long read	"TCPNio_readLong_before", "TCPNio_readLong_after"
int read	"TCPNio_readInt_before", "TCPNio_readInt_after"
accept	"TCPNio_accept_before", "TCPNio_accept_after"
Métodos RMI instrumentados	**Valores para atributo tipo**
invoke (servidor)	"RMI_beforeExecuting", "RMI_afterExecuting"
invoke (cliente)	"RMI_beforeInvoking", "RMI_afterInvoking"

5. Experimentos de Injeção de Falhas usando Comform

5.1. Experimento com Zorilla

Este experimento tem como propósito demonstrar que Comform pode, de fato, injetar adequadamente falhas de colapso em Zorilla [Drost et al. 2006], uma aplicação multi-protocolo baseada em UDP e TCP. A versão *1.0-beta1* de Zorilla, disponível para *download* [VRIJE UNIVERSITEIT 2007], foi utilizada. Interceptando o carregamento de classes dessa aplicação, constatou-se que ela é baseada nos protocolos TCP e UDP e faz uso tanto das implementações do pacote java.net como do pacote java.nio.

A situação emulada corresponde à apresentada na Figura 5, que indica qual máquina representa cada nodo no experimento. Na Figura, os nodos `dkw`, `jaguar` e `maverick` constituem uma rede Zorilla. O nodo `dkw` sofre uma emulação de colapso, com o uso de Comform, quando o nodo `mercedes` tenta conectar-se a `dkw` para realizar a submissão de um *job*. A seguir, o nodo `grantorino` tenta se unir à rede. Para a emulação correta do colapso, este deve ser percebido por todos os demais nodos do sistema. Como será mostrado, ao contrário de injetores de falhas voltados a aplicações baseadas em um único protocolo, Comform atinge esse objetivo, já que descarta tanto mensagens UDP como TCP no nodo `dkw` a partir do momento de ativação da falha.

Inicialmente, foi formada uma rede Zorilla, constituída pelos nodos `dkw`, `jaguar` e `maverick`. Para a inicialização de Zorilla nos nodos `jaguar` e `maverick`,

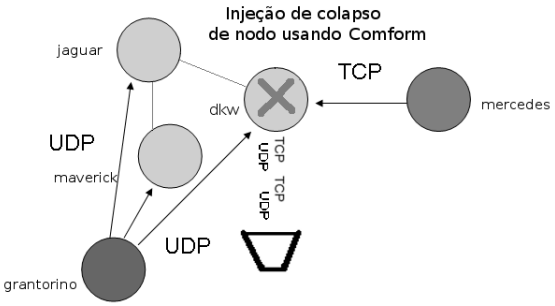


Figura 5. Planejamento de injeção de colapso em nodo Zorilla usando Comform.

foi utilizado, em cada máquina, o *script zorilla*, que é responsável pela inicialização e é encontrado no diretório *bin* de Zorilla *1.0-beta1*. Para a inicialização de Zorilla no nodo `dkw`, que é aquele onde Comform é executado com o propósito de injeção de uma falha de colapso, foi desenvolvido um novo *script* que alia as informações do *script zorilla* às informações necessárias para a instanciação de uma aplicação juntamente com Comform.

O *faultload* desenvolvido para o experimento, apresentado na Figura 6, ativa um colapso de nodo em `dkw` quando da tentativa de conexão feita por `mercedes` visando submeter um *job*. A falha de colapso é ativada no escopo do método `update`, que recebe como parâmetro as mensagens relacionadas aos protocolos sendo utilizados pela aplicação alvo. Quando uma mensagem TCP do tipo (atributo *type* de `TCPmessage`) "TCP_implAccept_after" proveniente de `mercedes` é recebida, a falha de colapso é ativada. O tipo "TCP_implAccept_after" foi utilizado porque, ao fim do corpo do método `implAccept`, o endereço do nodo que está tentando a conexão já é conhecido, de modo que é possível ativar a falha exatamente quando da tentativa de conexão por `mercedes`.

```
public class CrashFaultloadImplZorilla extends Faultload {
        CrashFault cf;
        int count = 0;
        public CrashFaultloadImplZorilla() throws Exception {
                cf = new CrashFault();
        }
        public void update(Message msg) throws Exception {
                if (msg.getClass().getName().equals("faultinjector.TCPmessage"))
                        if (((TCPmessage)msg).type.equals("TCP_implAccept_after") &&
                        (((TCPmessage)msg).remoteHostIpAddress).toString().equals("
                            /143.54.10.157")){
                                cf.activate();
        }
}
```

Figura 6. *Faultload* para emulação de colapso de nodo em `dkw`.

Depois de formada a rede Zorilla, uma tentativa de submissão de *job* no nodo `dkw` é realizada pelo nodo `mercedes` com o uso do *script zubmit*, responsável pela submissão de *jobs* a um nodo Zorilla (encontrado no diretório *bin* de Zorilla *1.0-beta1*). Para a tentativa de submissão, o nodo `mercedes` procura, primeiramente, estabelecer uma conexão com o nodo `dkw`. Porém, a falha de colapso é ativada em `dkw` quando desta tentativa e a exceção apresentada na Figura 7 é gerada em `mercedes`.

Logo após a ativação da falha, o nodo `grantorino` tenta se unir à rede. Para

```
ccmenegotto@mercedes:~/Desktop/apsteste/zorilla-1.0-beta1/bin$ ./zubmit -na
    143.54.10.205:5444 -c 1 .. ../../satin-2.1/examples/lib/satin-examples.jar
exception on running job: java.net.ConnectException: Connection setup failed
java.net.ConnectException: Connection setup failed
        at ibis.smartsockets.direct.DirectSocketFactory.createSingleSocket(
            DirectSocketFactory.java:1360)
        at ibis.smartsockets.direct.DirectSocketFactory.createSocket(DirectSocketFactory
            .java:1432)
        at ibis.smartsockets.direct.DirectSocketFactory.createSocket(DirectSocketFactory
            .java:1300)
        at ibis.zorilla.zoni.ZoniConnection.<init>(ZoniConnection.java:50)
        at ibis.zorilla.apps.Zubmit.main(Zubmit.java:278)
```

Figura 7. Exceção gerada no nodo `mercedes`.

tal, é executado o *script zorilla*. Inicialmente, o nodo apresenta um comportamento de inicialização convencional. A seguir, ele consegue se comunicar com os nodos `jaguar` e `maverick`, mas gera exceções ao tentar se comunicar com o nodo `dkw`, onde o colapso está sendo emulado. Os nodos `jaguar` e `maverick` também percebem o colapso de `dkw` e exceções do tipo `java.net.SocketTimeoutException` foram registradas em seus *logs* a cada tentativa de comunicação com `dkw`.

O experimento também foi realizado manualmente visando à comparação dos resultados com os obtidos na emulação com Comform. Para isso, a instrumentação do método `implAccept` foi alterada tal que, quando do recebimento de uma mensagem do tipo "TCP_implAccept_after" proveniente de `mercedes`, fosse executado um laço infinito. Esse laço infinito forneceu tempo necessário para remoção do cabo de força de `dkw` no contexto da execução do método `implAccept`, logo após a entrada do comando *zubmit* em um terminal de `mercedes`. A exceção obtida em `mercedes` foi igual à obtida na realização do experimento com Comform. Quanto a `jaguar` e `maverick`, o resultado obtido foi semelhante, mas, no lugar de exceções `java.net.SocketTimeoutException`, que haviam sido obtidas no experimento usando Comform, os experimentos manuais resultaram, em sua maioria, em algumas (de zero a 3) exceções `java.net.SocketTimeoutException` seguidas de exceções `java.net.NoRouteToHostException`. Já em `grantorino`, onde Zorilla foi instanciado somente após a ativação da falha, só foram registradas exceções `java.net.NoRouteToHostException` nos experimentos manuais. Porém, apesar do nome diferente dessa nova exceção registrada nos experimentos manuais, observou-se que todas as demais informações, referentes à origem da exceção, são iguais às obtidas no experimento com Comform, de modo que o colapso é emulado com precisão adequada. Uma investigação, baseada em outro experimento, no qual o colapso em `dkw` foi emulado utilizando-se regras do IPTables responsáveis pelo descarte de todos os pacotes recebidos e enviados sem que isso resultasse na geração desse tipo de exceção em `mercedes`, levou à conclusão de que a geração de exceções `NoRouteToHostException` só seria viável emulando-se falhas em nível de abstração mais baixo.

A análise dos *logs* do experimento com Comform mostrou que as seguintes classes utilizadas por Zorilla para comunicação UDP ou TCP foram instrumentadas no nodo `dkw`: `ServerSocket`, `DatagramSocket`, `Socket`, `SocketInputStream` e `SocketOutputStream` de `java.net` e `DatagramChannelImpl` de `sun.nio.ch`. Todos os nodos envolvidos no experimento perceberam o colapso

de dkw e a falha foi emulada consistentemente. O mesmo não ocorreria caso um injetor de falhas voltado ao teste de aplicações Java baseadas em um único protocolo tivesse sido utilizado. Não foram encontrados registros de injetores de falhas que tratem da instrumentação do pacote java.nio, de modo que Comform é pioneiro nesse sentido.

5.2. Experimento com Aplicação RMI

Este experimento mostra a habilidade de Comform para injetar falhas em testes de caixa branca (considerando o código fonte da aplicação alvo), além da correção da estratégia empregada para emulação de colapsos de nodo. Ele foi conduzido em dois computadores, mercedes e dkw. A aplicação alvo inclui um servidor que implementa um interface remota composta de dois métodos. O primeiro, multiply, multiplica duas matrizes recebidas como parâmetro. O segundo, sum, soma duas matrizes recebidas como parâmetro. A aplicação também inclui um cliente que obtém uma referência remota, s, ao servidor visando ser capaz de invocar métodos no servidor. Três matrizes – a, b e c – são declaradas no cliente. A Figura 8 mostra a ordem de invocação dos métodos remotos.

```
int [][] d = s.multiply(a, b);
d = s.multiply(a, c);
d = s.sum(a, b);
```

Figura 8. Um trecho de código do cliente.

O servidor foi executado em dkw, com Comform, e o cliente em mercedes. Para ilustrar a habilidade de Comform para injetar falhas em pontos específicos da execução de uma aplicação, um colapso de nodo foi injetado em dkw antes da segunda invocação de multiply, ou seja, foi emulado o colapso do servidor depois que o cliente invocou o método, mas antes que o servidor retornasse o resultado. Para comparação, o experimento também foi realizado manualmente pela substituição do código de multiply por um laço infinito, que proveu tempo suficiente para consequente remoção do cabo de força do nodo servidor enquanto ele executava o laço. A Figura 9 mostra o *faultload* desenvolvido para esse teste. A classe CrashFaultload estende a classe Faultload da API de Comform. A falha é ativada no escopo do método update, que recebe como parâmetro mensagens relacionadas aos protocolos em uso.

```
public class CrashFaultload extends Faultload {
  private CrashFault cf;
  int count = 0;
  public CrashFaultload() throws Exception {
    cf = new CrashFault();}
  public void update(Message msg) throws Exception {
    if (msg.getClass().getName().equals("faultinjector.RMIRequest")){
      if (((RMIRequest)msg).method.getName().equals("multiply") && ((RMIRequest) msg).
          type.equals("RMI_beforeExecuting")) {
        count++;
        if (count==2) cf.activate();}}}
}
```

Figura 9. *Faultload* para emulação de colapso.

A situação emulada deixa o cliente esperando que o servidor processe a requisição. Como a aplicação não é tolerante a falhas, quando um colapso é injetado, o cliente não é

capaz de saber se o servidor ainda está processando a requisição ou se ele sofreu colapso. Após a expiração do *timeout* do TCP, que pode levar um longo tempo dependendo da configuração do cliente, uma exceção é disparada na aplicação cliente indicando que a conexão sofreu *timeout*. Adicionalmente, exatamente a mesma exceção foi capturada tanto no experimento manual como no realizado com Comform, mostrando a adequação da abordagem usada para emulação de colapsos de nodo. A Figura 10 mostra um trecho da mensagem de exceção. A realização de um experimento como esse seria inviável com a utilização de um injetor de falhas voltado ao teste de protocolos como FIRMAMENT.

```
Exception in thread "main" java.rmi.UnmarshalException: Error unmarshaling return header
    ; nested exception is:
    java.net.SocketException: Connection timed out
    at sun.rmi.transport.StreamRemoteCall.executeCall(StreamRemoteCall.java:209)
    at sun.rmi.server.UnicastRef.invoke(UnicastRef.java:142)
    at java.rmi.server.RemoteObjectInvocationHandler.invokeRemoteMethod(
        RemoteObjectInvocationHandler.java:178)
    at java.rmi.server.RemoteObjectInvocationHandler.invoke(
        RemoteObjectInvocationHandler.java:132)
    at $Proxy0.multiply(Unknown Source)
    at Client.main(Client.java:47)
Caused by: java.net.SocketException: Connection timed out
```

Figura 10. Exceção gerada no cliente.

6. Considerações Finais

Este artigo mostrou que muitos injetores de falhas de comunicação encontrados na literatura não são capazes de testar aplicações Java multiprotocolo e que outros possuem potencial para o teste dessas aplicações, mas impõem grandes dificuldades aos engenheiros de testes. Com base na necessidade identificada, apresentou a proposta de Comform. Quanto à capacidade de emular os tipos de falhas de comunicação que podem ocorrer comumente em ambientes de rede, Comform já é superior a ferramentas como FAIL-FCI [Hoarau et al. 2007], mas ainda pode ser melhorado pela inclusão de novos tipos de falhas em sua arquitetura. Quanto à preservação do código fonte da aplicação alvo, ela é atingida com o uso da combinação do pacote `java.lang.instrument` e de Javassist. A capacidade de injetar falhas tanto independentemente do conhecimento do código fonte da aplicação alvo como também o levando em consideração foi mostrada nos experimentos. Quanto ao provimento de um mecanismo adequado para descrição de Módulos de Carga de Falhas, a estratégia empregada não pôde ser tratada com maiores detalhes neste artigo, mas os experimentos mostraram sua facilidade de uso.

Trabalhos futuros incluem a continuação do desenvolvimento de Comform visando tratar mais tipos de falhas e protocolos. Seria interessante implementar falhas de *omissão* e investigar alternativas para emulação de particionamento de rede. Quanto a outros protocolos, seria útil tratar outros largamente utilizados que sejam baseados em UDP ou TCP. Comform já tem essa capacidade para o caso de *testes de caixa preta*. Para *testes de caixa branca*, ele pode ser estendido para prover facilidades para ativação de falhas como as que já oferece para teste de caixa branca de aplicações Java que usam RMI.

Referências

Ban, B. (2002). JGroups - a toolkit for reliable multicast communication. 2002. Disponível em: <http://www.jgroups.org>. Acesso em: jan. 2009.

Cézane, D. et al. (2009). Um injetor de falhas para a avaliação de aplicações distribuídas baseadas no commune. In *Anais do Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos, SBRC, 27.*, volume 1, pages 901–914, Recife. SBC.

Chandra, R. et al. (2004). A global-state-triggered fault injector for distributed system evaluation. *IEEE Transactions on Parallel and Distributed Systems*, 15(7):593–605.

Chiba, S. (1998). Javassist. 1998. Disponível em: <http://www.csg.is.titech.ac.jp/~chiba/javassist/>. Acesso em: jan. 2009.

Drebes, R. J. (2005). Firmament: um módulo de injeção de falhas de comunicação para linux. 2005. 87 f. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

Drost, N., van Nieuwpoort, R. V., and Bal, H. (2006). Simple locality-aware co-allocation in peer-to-peer supercomputing. In *Proc. of the 6th International Symposium on Cluster Computing and the Grid Workshops, CCGRIDW*, volume 2, Singapore. IEEE.

Gerchman, J. and Weber, T. S. (2006). Emulando o comportamento de tcp/ip em um ambiente com falhas para teste de aplicações de rede. In *Anais do Workshop de testes e tolerância a falhas, WTF, 7.*, volume 1, pages 41–54, Curitiba. SBC.

Hoarau, W., Tixeuil, S., and Vauchelles, F. (2007). FAIL-FCI: Versatile fault injection. *Future Generation Computer Systems*, 23(7):913–919.

Jacques-Silva, G. et al. (2006). A network-level distributed fault injector for experimental validation of dependable distributed systems. In *Proc. of the 30th Int. Computer Software and Applications Conference, COMPSAC*, pages 421–428, Chicago. IEEE.

Looker, N., Munro, M., and Xu, J. (2004). Ws-fit: a tool for dependability analysis of web services. In *Proc. of the 28th Annual International Computer Software and Applications Conference, COMPSAC*, volume 2, pages 120–123, Hong Kong. IEEE.

Murray, P. (2005). A distributed state monitoring service for adaptive application management. In *Proc. of the International Conference on Dependable Systems and Networks, DSN*, pages 200–205, Yokohama. IEEE.

Pezzé, M. and Young, M. (2008). *Software Testing and Analysis: Process, Principles, and Techniques*. John Wiley & Sons.

Stott, D. T. et al. (2000). Nftape: A framework for assessing dependability in distributed systems with lightweight fault injectors. In *Proc. of the 4th International Computer Performance and Dependability Symposium, IPDS*, pages 91–100, Chicago. IEEE.

SUN MICROSYSTEMS (2008). Java platform standard ed. 6. 2008. Disponível em: <http://java.sun.com/javase/6/docs/api/>. Acesso em: jan. 2009.

Vacaro, J. C. (2007). Avaliação de dependabilidade de aplicações distribuídas baseadas em rmi através de injeção de falhas. 2007. 89 f. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

VRIJE UNIVERSITEIT (2007). Ibis: Grids as promised. 2007. Disponível em: <http://www.cs.vu.nl/ibis/downloads.html>. Acesso em: out. 2009.

XI Workshop de Testes e Tolerância a Falhas

♦

Sessão Técnica 3

Testes e Sistemas Embarcados

Um Framework de Geração de Dados de Teste para Critérios Estruturais Baseados em Código Objeto Java

Lucilia Yoshie Araki[1], Silvia Regina Vergilio[1]

[1]Departamento de Informática – Universidade Federal do Paraná (UFPR)
Caixa Postal 19.081 – 81531 - 970 – Curitiba – PR

{lya_araki@yahoo.com.br, silvia@inf.ufpr.br}

Resumo. *O teste evolutivo de software orientado a objeto é uma área de pesquisa emergente. Algumas abordagens promissoras sobre o assunto são encontradas na literatura, entretanto, estas não consideram critérios propostos recentemente que utilizam o código objeto Java para obter os requisitos de teste. Além disso, os trabalhos geralmente não estão integrados a uma ferramenta de teste. Neste artigo, um framework, chamado TDSGen/OO para geração de dados de teste é descrito. TDSGen/OO utiliza Algoritmos Genéticos e trabalha de maneira integrada com a ferramenta JaBUTi, que implementa diferentes critérios de teste baseados no bytecode e em mecanismos de tratamento de exceções, permitindo o teste de componentes mesmo que o código fonte não esteja disponível. Alguns resultados preliminares são também apresentados que mostram benefícios no uso do framework.*

Abstract. *The evolutionary test of object-oriented software is an emergent research area. We find in the literature promising approaches on this subject, however, those approaches do not consider some recent test criteria that use the Java Byte-code to derive the test requirements. In addition to this, they are not usually integrated to a test tool. In this paper, we describe a framework, named TDSGen/OO to test data generation. The framework uses a Genetic Algorithm and is integrated with JaBUTi, a tool that implements different test criteria based on bytecode and exception-handling mechanisms, allowing the test of components even if the source code is not available. Some preliminary evaluation results are also presented.*

1. Introdução

Nos últimos anos, o paradigma de orientação a objeto ganhou importância e vem sendo intensivamente utilizado. O uso de recursos específicos deste paradigma pode introduzir novos e diferentes tipos de defeitos. Por isso, a aplicação de um critério de teste neste contexto é fundamental.

Diferentes critérios foram propostos. Eles utilizam diferentes tipos de informação para derivar os requisitos de teste. Muitos critérios são baseados na especificação [13] e em diversos modelos, como diagrama de estados, casos de uso, classes, e etc. Outros critérios são baseados no programa. Os critérios estruturais são

geralmente baseados nos grafos de fluxo de controle e interações de fluxo de dados, considerando sequências de chamadas de métodos [6].

Recentemente, foram propostos alguns critérios estruturais de teste que consideram informações do código objeto Java [21]. Estes critérios podem ser aplicados mesmo quando o código fonte não está disponível. Isso é bastante comum na maioria dos testes de componentes de software, o que torna um critério baseado em bytecode muito útil. Além disso, uma ferramenta, chamada JaBUTi (Java Bytecode Understanding and Testing) [20, 21], está disponível para aplicação de tais critérios. JaBUTi implementa critérios baseados em fluxo de controle e em fluxo de dados considerando mecanismo de manipulação de exceções.

A ferramenta tem como objetivo o teste de unidade de uma determinada classe. Gera os elementos requeridos pelos critérios de aplicação, mostra o gráfico correspondente, e produz relatórios de cobertura com respeito ao conjunto de teste fornecido pelo testador, que é responsável pela geração manual dos dados de teste para cobrir um critério de teste específico. Esta tarefa torna o teste demorado, difícil e caro e, em geral, é feita manualmente.

Esta limitação é tema de pesquisa da área denominada Teste Evolucionário [25], que aplica algoritmos evolutivos para geração de dados de teste. A maioria dos trabalhos em teste evolucionário, no entanto, destinam-se ao teste de unidade do código procedural [1,9,10,11]. Podemos citar, entre estes trabalhos, a ferramenta TDSGen [3], que gera dados de teste para critérios estruturais e critérios baseados em defeito, para programas em C. TSDGen implementa mecanismos de hibridização para melhorar o desempenho do algoritmo genético, e trabalha com duas ferramentas de teste.

No contexto de software orientado a objeto, o teste evolucionário é considerado uma área emergente de pesquisa e ainda não foi investigado adequadamente [15]. Há um número reduzido de trabalhos. Tonella [19] foi o primeiro a investigar o teste evolutivo de classes usando Algoritmos Genéticos (GA) para satisfazer o critério estrutural todos-os-ramos. Outros trabalhos investigam outras técnicas como: Colônia de Formigas [8], Algoritmos Evolutivos Universais [22], Programação Genética [18.24], Algoritmos de Distribuição de Estimativas [17], e etc. [23]. Estes trabalhos usam diferentes técnicas para diferentes finalidades. A maioria deles destina-se apenas a um critério de teste, geralmente o critério todos-nós ou todos-ramos. Eles não consideram os critérios baseados em bytecode, e também não oferecem uma implementação integrada com uma ferramenta que apóie o critério escolhido.

Devido a isso, neste trabalho, um framework para geração de dados de teste é apresentado. O framework, chamado TDSGen/OO (Test Data Set Generator for OO Software), tem como finalidade a geração de dados de teste para satisfazer os critérios baseados em fluxo de controle e em fluxo de dados que consideram o bytecode, implementados pela ferramenta de teste JaBUTi. O framework implementa um algoritmo genético e alguns mecanismos para melhorar o desempenho que foram utilizados com sucesso por TDSGEN [3] no teste de software procedural.

A idéia é fornecer um ambiente para a aplicação de todos os critérios executados em uma estratégia de teste, e reduzir o esforço e o custo do teste no contexto de software orientado ao objeto.

As demais seções deste artigo estão organizadas da seguinte forma. A Seção 2 discute trabalhos relacionados. Seção 3 descreve o framework TDSGen/OO. Resultados de um estudo preliminar são apresentados na Seção 4. As conclusões e trabalhos futuros estão na Seção 5.

2. Trabalhos Relacionados

Existem vários trabalhos sobre teste evolucionário de programas procedurais que podem ser encontrados na literatura, [1, 5, 9, 10]. As principais técnicas utilizadas por estes trabalhos para a geração de dados de teste são: Hill Climbing, Simulated Annealing, Algoritmos Genéticos, Programação Genética, e etc. Além das técnicas, outra diferença está na função de aptidão (ou fitness) usada. Alguns trabalhos usam uma função de fitness orientada à cobertura de um critério. Nesse caso, são abordados diferentes critérios, com base no: fluxo de controle e de dados, ou defeitos e teste de mutação. Outras funções têm o objetivo de cobrir um elemento específico exigido.

Por outro lado, no contexto de orientação a objeto, um número menor de trabalhos é encontrado. Poucas técnicas foram exploradas, bem como, poucos critérios e funções de fitness.

O primeiro trabalho a explorar o teste de software orientado a objeto foi a abordagem baseada em Algoritmos Genéticos, proposta por Tonella [19]. A abordagem propôs uma representação baseada em gramáticas (Figura 1) para os dados de teste, capaz de representar seqüência de invocações de métodos, além da simples representação das entradas utilizadas nos programas tradicionais.

```
<chromosome>   ::= <actions> @ <values>

<actions>   ::= <action> {: <actions>} ? <action>

::=  $id=constructor({<parameters>}? )

| $id = class # null

| $id . method( {<parameters>}? )

<parameters>::=  <parameter> {, <parameters>? }

<parameter>      ::=   builtin-type {<generator>} ? | $id

<generator>      ::=  [low ;  up] | [genClass]

<values>         ::= <value> {, <value>} ?

<value>          ::= integer | real |  boolean | string
```

Figura 1: Gramática introduzida por Tonella (extraída de [19]).

A gramática representa um dado de teste (ou cenário de teste) para software orientado a objeto. O "@" divide o cromossomo em duas partes. A primeira parte representa a seqüência de invocações de métodos ou construtores. A segunda parte contém os valores de entrada para essas operações, separadas por “,”. Uma ação pode representar um novo objeto $id ou uma chamada para um método identificado por $id.

Os parâmetros podem representar tipos tais como inteiro, real, string ou booleanos. O trabalho também propôs um conjunto de operadores evolutivos de cruzamento (crossover) e mutação. A função de fitness é orientada a um objetivo.

O trabalho de Liu et al. [8] utiliza um algoritmo de Colônia de Formigas para gerar a menor seqüência de invocações de métodos. O objetivo é cobrir as arestas em métodos privados e protegidos.

Wappler e Lammermann [22] apresentam uma abordagem baseada em Algoritmos Evolutivos Universais para permitir a aplicação de diferentes algoritmos de busca, tais como Hill Climbing e Simulated Annealing. A função de fitness usa mecanismos para penalizar seqüências inválidas e guiar a busca, pois sua representação pode gerar indivíduos que poderiam ser transformados em programas incorretos. Um trabalho posterior de Wappler e Wegner [24] não apresenta esse problema e codifica possíveis soluções usando Programação Genética Fortemente Tipada. As seqüências são representadas por árvores, e resultados promissores foram obtidos. O uso de Programação Genética também foi explorado por Seesing e Gross [18].

Sagarna et al. [17] apresentam uma abordagem baseada em Algoritmos de Distribuição de Estimativas (EDA) para gerar dados de teste para a cobertura de um ramo no teste de unidade de programas Java.

Um problema com os trabalhos mencionados é que a representação dos dados de teste, assim como a abstração do fluxo de execução e/ou de dados estão baseadas no código fonte. Se o código não está disponível, os algoritmos não podem ser aplicados. Para superar esta limitação os trabalhos mais recentes tentam gerar dados de teste analisando diretamente o bytecode. O trabalho de Cheon et al. [2] propõe extensões ao compilador JML para produzir informações sobre cobertura e utiliza as especificações JML para determinar o resultado do teste. O trabalho descrito em [12] gera dados de teste resolvendo restrições. Estes trabalhos, entretanto, não usam algoritmos evolutivos, baseados em busca meta-heurística.

Os trabalhos relatados em [15,16] abordam o teste evolutivo considerando bytecode. Os trabalhos utilizam um pacote que inclui diferentes Algoritmos Evolutivos Universais e Programação Genética Fortemente Tipada, de forma semelhante ao trabalho descrito em [22]. A geração é guiada por informações extraídas do bytecode.

Na próxima seção, é descrito o framework TSDGen/OO para a geração de dados de teste no contexto de software orientado a objeto, integrado com a ferramenta de teste JaBUTi. A representação do indivíduo é baseada no trabalho de Tonella [19], mas diferentemente, a ferramenta executa uma função de fitness orientada à cobertura dos critérios estruturais baseados em fluxo de controle e em fluxo de dados, aplicados a um modelo extraído do bytecode. Um outro aspecto que torna o trabalho diferente dos demais trabalhos que também consideram o bytecode na geração [15.16] é a função implementada que permite uma melhoria no desempenho de um algoritmo genético simples, considerando trabalhos anteriores [3] e toda a experiência adquirida com programas tradicionais.

3. Ferramenta TDSGen/OO

TDSGen/OO gera dados de teste para os critérios estruturais implementados pela JaBUTi, que são critérios baseados em fluxo de controle e em fluxo de dados. A Figura 2 apresenta sua estrutura, baseada no trabalho relatado em [3]. TDSGen/OO contém quatro módulos principais: Geração da População, Avaliação, Evolução e Início. A informação produzida transmitida através dos módulos está representada nesta figura por elipses. Na sequência é apresentada uma breve descrição de cada módulo.

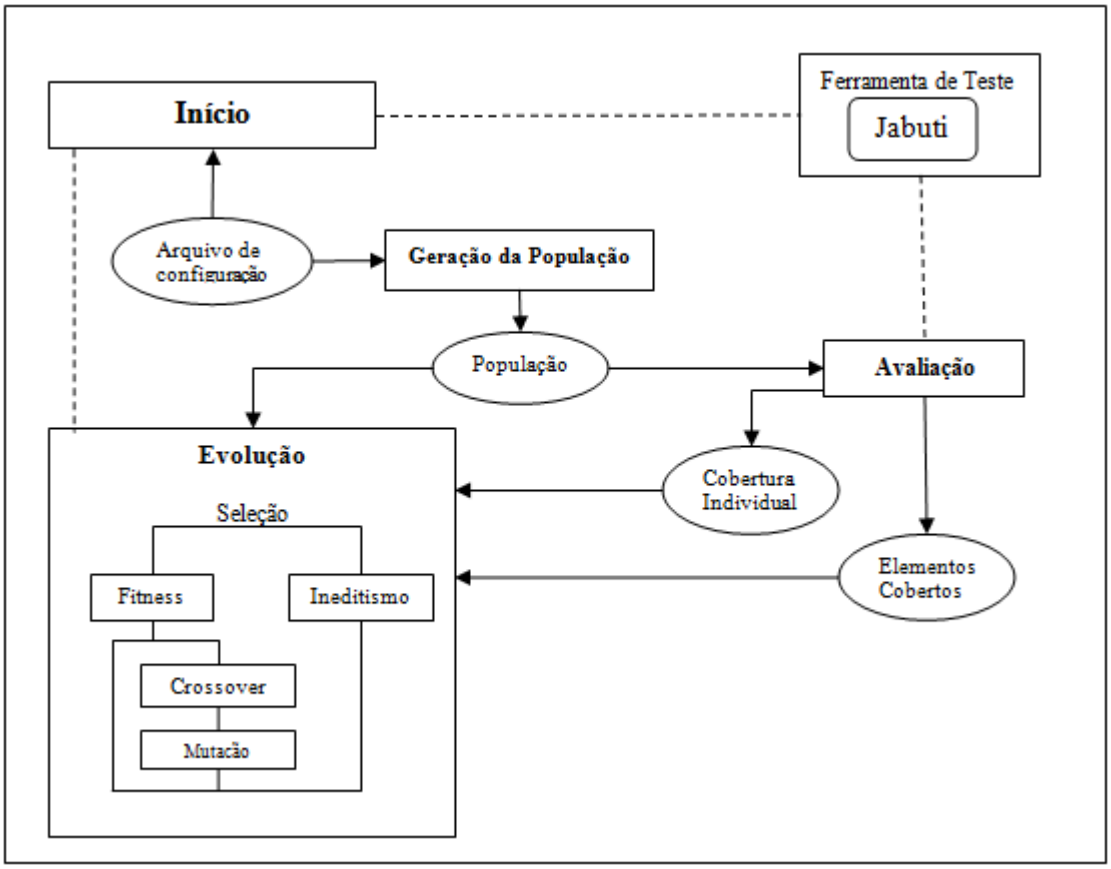


Figura 2: Principais módulos do framework TDSGen/OO

A. Início

O Módulo Início é responsável por receber a configuração inicial (arquivo de configuração) do testador e por controlar os outros módulos. O testador pode também usar uma interface gráfica para fornecer as informações iniciais.

Uma configuração possui duas seções, sendo que uma seção - ferramenta de teste - inclui parâmetros para a JaBUTi: o nome do arquivo fonte que contém a classe a ser testada, e o critério escolhido; a outra seção - estratégias de evolução - está relacionada com o processo de evolução e será utilizada pelo Módulo Evolução, contendo taxa de crossover, taxa de mutação, tamanho da população, número máximo de gerações, método de seleção (torneio[1] ou roleta[2]), elitismo e ineditismo.

[1] Seleção por Torneio: um número p de indivíduos da população é escolhido aleatoriamente para formar uma sub-população temporária. Deste grupo, é selecionado o melhor indivíduo.

Um exemplo de configuração é apresentado na Figura 3. A população tem 50 indivíduos, ou seja, 50 candidados à solução do problema, com um tamanho máximo, relacionado ao número de invocações de métodos igual a 10. Elitismo e ineditismo não estão habilitados. As taxas de crossover são 0,75 e de mutação 0,01. No crossover dois indivíduos pais são selecionados e seu material genético é combinado, permutando uma parte de um dos pais por uma parte do outro, gerando um novo indivíduo. O operador de mutação modifica aleatoriamente um ou mais invocações de métodos do indivíduo.

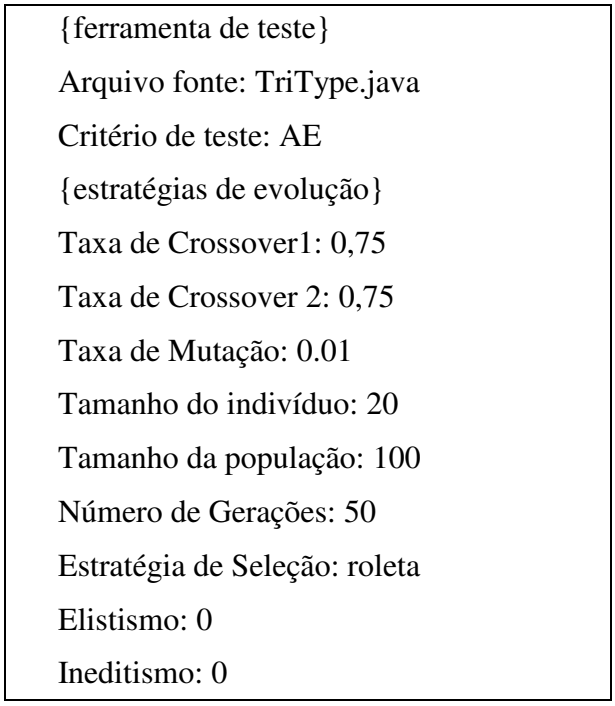


Figura 3: Configuração básica usada pela TDSGen/OO

B. Geração da População

Este módulo trabalha de acordo com as informações contidas no arquivo de configuração. Ele gera a população inicial e também é responsável pela codificação e decodificação de cada indivíduo.

A representação escolhida para o cromossomo (indivíduo) usa a gramática introduzida por Tonella [19] apresentada na Seção 2. O módulo garante que os cromossomos sejam bem-formados, ou seja, contenham para cada chamada de método seu parâmetro correspondente; $id é sempre determinado antes da sua utilização, e sempre é associado a uma ação correspondente.

A população inicial é gerada aleatoriamente, sendo esta população um conjunto de possíveis dados de teste. A Figura 4 apresenta um exemplo da população com 3 indivíduos para o programa Trityp, que verifica se as suas três entradas compostas por números inteiros formam um triângulo. Em caso afirmativo, o programa imprime o tipo

---

[2] Seleção por Roleta: Especifica a probabilidade de que cada indivíduo seja selecionado para a próxima geração. Cada indivíduo da população recebe uma porção da roleta proporcional a sua probabilidade.

de triângulo formado. A primeira parte identificada por $ representa as chamadas de métodos do programa TriTyp e logo após o @ encontram-se os parâmetros necessários para cada chamada de método.

```
$t=TriTyp() : $t.setI(int)  : $t.setJ(int) : $t.setK(int) : $t.type() @ 5, 5, 5
$t=TriTyp() : $t.setI(int) : $t.setJ(int) :$t.setK(int) : $t.type() @ 2, 3, 2
$t=TriTyp() :  $t.setI(int) : $t.setJ(int) : $t.setK(int) : $t.type() @ 3, 6, 7
```

Figura 4: População gerada.

C. Ferramenta de Teste

A ferramenta JaBUTi [20,21] utiliza o bytecode como base para construir o grafo de fluxo de controle (GFC). A ferramenta distingue as instruções que são cobertas sob execução normal do programa de outros que exigem uma exceção para serem executadas, e por causa disto, o critério todos-nós foi subdividido em dois critérios de teste não-sobrepostos, que são:

- Critério todos-nós independentes de exceção (all-nodes$_{ei}$): requer a cobertura de todos os nós do GFC não relacionados ao tratamento de exceção.
- Critério todos-nós dependentes de exceção (all-nodes$_{ed}$): requer a cobertura de todos os nós do GFC relacionados ao tratamento de exceção.

Analogamente, outros critérios são também subdivididos: critérios todos-ramos independentes de exceção (all-edges$_{ei}$) e todos-ramos dependentes de execeção (all-edges$_{ed}$); todos-usos independentes de exceção (all-uses$_{ei}$) , todos-usos dependentes de exceção (all-uses$_{ed}$), e assim por diante.

D. Avaliação

O módulo Avaliação verifica a cobertura de cada indivíduo (número de elementos cobertos pelo critério especificado no arquivo de configuração) usando o módulo de avaliação da JaBUTi. Como mencionado anteriormente, cada indivíduo é convertido em uma entrada (um arquivo JUnit) para o programa que está sendo testado pelo Módulo Avaliação. A lista dos elementos cobertos também é salva em um arquivo, pois a análise feita pela JaBUTi pode ficar muito demorada. A matriz apresentada na Figura 5 é obtida e utilizada pela função de fitness que será explicada a seguir.

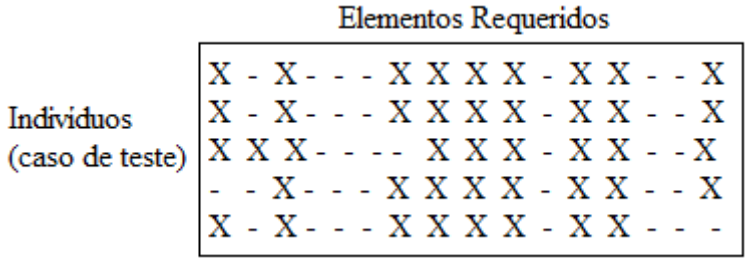


Figura 5: Informação utilizada para avaliação do fitness.

E. Evolução

O Módulo Evolução é responsável pelo processo de evolução e pela aplicação dos operadores genéticos. Os operadores foram implementados com base no trabalho de Tonella [19]. Sendo:

- Mutação: este operador faz alterações, inserções ou remove entradas, construtores, objetos e chamadas de método.
- Crossover: neste operador o ponto de corte pode ser de ambas as partes dos cromossomos, na seqüência de invocações dos métodos ou na parte que contém os parâmetros.

O processo de evolução termina quando o número de gerações é atingido. Alguns mecanismos são utilizados com o objetivo de aumentar o desempenho. Esses mecanismos são ativados pelo testador no arquivo de configuração.

- Fitness: A aptidão de um indivíduo é calculada com base na matriz da Figura 5 e é dada por:

$$\text{Fitness} = \frac{\text{número\_elementos\_cobertos}}{\text{número\_elementos\_requeridos}}$$

Com base na aptidão, os indivíduos são selecionados e os operadores genéticos são aplicados. Nesta versão, TDSGen/OO implementa duas estratégias de seleção: roleta e torneio. As taxas são passadas através do arquivo de configuração.

- Elitismo: Esta estratégia introduz indivíduos na próxima geração com base em uma lista ordenada pelo valor de fitness. Isso garante que os indivíduos com valor de fitness alto estejam na nova população.
- Ineditismo: Esta estratégia tem o objetivo de reduzir o número de indivíduos semelhantes na população (com base na estratégia sharing [4]). Para introduzir os indivíduos na próxima geração, é considerada uma lista ordenada pela métrica de ineditismo. É dado um bônus ao indivíduo que cobre um elemento requerido, não coberto por outros indivíduos da população. Cada elemento i exigido é associado a um bônus de acordo com o número de indivíduos que o cobrem.

$$Bonus_i = 100 \times \left(1 - \frac{nro\_indivíduos\_cobrem}{nro\_total\_individuos}\right)$$

Cada indivíduo x recebe um bônus de ineditismo, que é dado pela soma do bônus para x de cada elemento requerido i.

$$BonusIneditismo_i = \sum_{i=0}^{número\ de\ elementos\ requeridos} Bonus_i$$

4. Resultados Experimentais

Para avaliar a ferramenta TDSGen/OO, foi realizado um estudo com quatro programas em Java: TriTyp, Bub, Bisect e Mid. Esses programas foram utilizados por outros autores [14]. Eles são simples, mas podem fornecer uma idéia inicial com relação ao uso da ferramenta.

Foram avaliadas três estratégias, utilizando TDSGen/OO.

a) Geração aleatória (estratégia RA): os dados de teste são obtidos usando a população inicial gerada pelo módulo Geração da População, atribuindo o valor zero ao parâmetro: número de gerações;

b) Geração baseada em Algoritmo Genético (estratégia GA): os dados de teste são obtidos pela desativação dos parâmetros elitismo e ineditismo.

c) Geração baseada na estratégia Ineditismo (estratégia GAU): os dados de teste são obtidos ativando os parâmetros elitismo e ineditismo.

Os critérios todos-ramos$_{ei}$ (AE) e todos-usos$_{ei}$ (AU), implementados pela JaBUTi, foram escolhidos neste estudo, representando, respectivamente, a categoria de critérios baseado em fluxo controle e fluxo de dados.

Os parâmetros elitismo e ineditismo da seção evolução são diferentes de acordo com a estratégia, como explicado anteriormente. Os outros parâmetros foram experimentalmente fixados, exceto o tamanho da população (Figura 6).

```
{estratégias de evolução}
Taxa de Crossover1: 0.75
Taxa de Crossover 2: 0.75
Taxa de Mutação: 0.75
Tamanho do indivíduo: 20
Tamanho da população: 100
Número de Gerações: 50
Estratégia de Seleção: roleta
```

Figura 6: Configuração utilizada no experimento

Para todos os critérios e estratégias, foram realizadas 5 execuções um valor médio para a cobertura foi obtido. Os resultados são apresentados na Tabela 1.

Tabela 1: Cobertura obtida (em porcentagem).

	RA		*GA*		*GAU*	
	AE	***AU***	***AE***	***AU***	***AE***	***AU***
1-TriTyp	44	39	45	40	56	43
2-Bub	35	26	31	18	75	60
3-Mid	65	73	58	57	88	96
4-Bisect	30	33	31	40	53	60

A. Análise dos Resultados

Para uma melhor avaliação, os resultados de cobertura são apresentados em gráficos nas Figuras 7 e 8. Se a cobertura de todos os programas for considerada, as estratégias RA e GA apresentam uma cobertura média semelhante, independentemente do critério. RA apresenta melhor desempenho para os programas Bub e Mid, e GA para os programas TriTpy e Bisect. Nestes casos, a estratégia GA apresenta uma menor cobertura, pois alguns bons indivíduos iniciais são perdidos durante o processo de evolução.

Esse problema não acontece com a estratégia GAU. Aqueles bons indivíduos recebem o bônus ineditismo, e não são descartados facilmente.

Uma explicação para o comportamento da estratégia GA é o menor número de gerações utilizadas (50). Talvez o algoritmo genético pudesse voltar às boas soluções com um número maior de gerações. Mas isso demonstra que o uso do ineditismo realmente contribui para melhorar o desempenho e diminui os custos de execução do Algoritmo Genético.

Outro ponto é que a estratégia GAU sempre apresenta a maior cobertura média, com uma média de 20% de melhora para ambos os critérios.

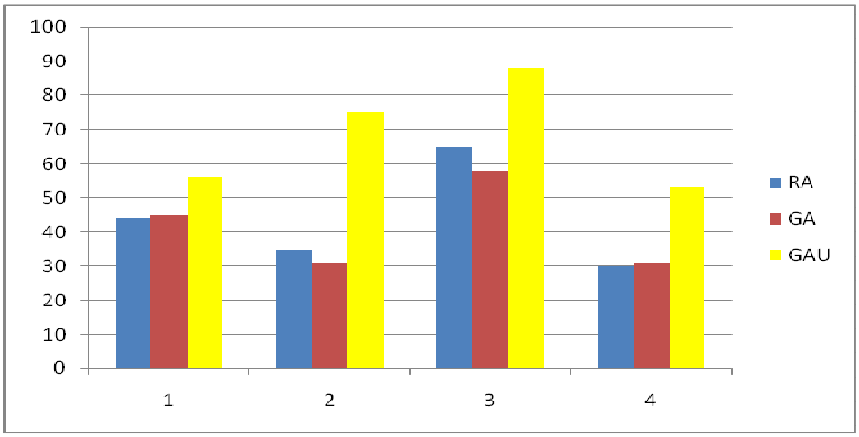


Figura 7: Comparação de estratégias – Cobertura critério AE.

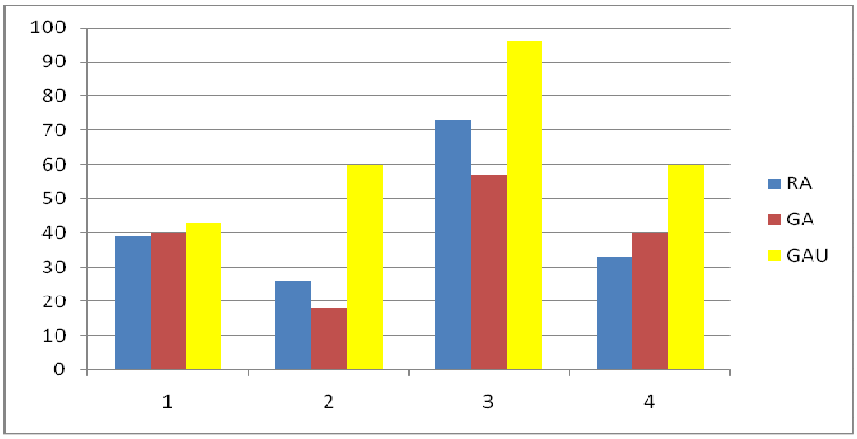


Figura 8: Comparação de estratégias – Cobertura critério AU

Observa-se que a cobertura média de AE é muito semelhante à cobertura do critério AU. Ela é maior para os programas Bisect e Bub. Isso é inesperado, pois para satisfazer um critério baseado em fluxo de dados geralmente é mais difícil do que um critério baseado em fluxo de controle. Observa-se também que não há diferença significativa entre os critérios. Este fato pode ser devido ao tamanho dos programas, que são pequenos.

Com relação ao tempo de execução dos algoritmos, a estratégia de RA é a menos custosa. Não há diferença significativa entre as outras estratégias GA e GAU. As médias dos tempos de execução destas estratégias são muito semelhantes, e cerca de 2 vezes o tempo de execução RA.

5. Conclusões

Neste trabalho, o framework TDSGen/OO é descrito. TDSGen/OO implementa um Algoritmo Genético para gerar dados de teste no contexto de software orientado a objeto.

O framework tem algumas características importantes que o tornam diferente de outros trabalhos encontrados na literatura. TDSGen/OO trabalha de forma integrada com a ferramenta de teste de JaBUTi, uma ferramenta que permite a aplicação de critérios baseados em fluxo de controle e de dados no teste de unidade de classes Java. Uma característica importante destes critérios é obter os requisitos de teste com base no bytecode e mecanismos de tratamento de exceção.

Desta forma, TDSGen/OO pode ser aplicada mesmo se o código fonte não estiver disponível, pois a função de fitness implementada baseia-se na cobertura dos critérios, fornecida pela ferramenta de teste. Não é necessária qualquer análise adicional ou interpretação do programa em teste.

O fato de estar integrado com a ferramenta JaBUTi permite que o framework seja utilizado em uma estratégia, incluindo diferentes e complementares critérios de teste.

Além disso, com base em um trabalho anterior sobre a geração de dados de teste em código procedural, TDSGen/OO implementa uma estratégia baseada na métrica de

ineditismo, o que parece ser fundamental nos casos em que apenas um caminho particular ou caso de teste cobre um elemento requerido, para manter este teste na população.

No estudo preliminar conduzido, a estratégia GAU (GA + ineditismo) obteve a maior cobertura, sem aumentar o custo. No entanto, outros estudos experimentais devem ser conduzidos. Algumas mudanças (aumento de valor) no número máximo de gerações foram testadas e foi observado um aumento na cobertura para as estratégias GA e GAU. Outro parâmetro que pode influenciar na cobertura obtida de todas as estratégias, incluindo a estratégia aleatória é o tamanho do indivíduo e da população.

Novos experimentos devem avaliar: os parâmetros da seção de avaliação, tais como o número de gerações e aplicação dos operadores genéticos; a eficácia da geração de dados de teste e os custos do algoritmo considerando programas maiores e reais.

Uma limitação observada é a dificuldade de alcançar uma cobertura completa. de A cobertura completa nem sempre é possível devido a elementos não executáveis. Portanto pretende-se incorporar novas funcionalidades ao framework TDSGen/OO para reduzir essas limitações, mas a participação do testador, sempre será necessária.

Um outro trabalho que está sendo realizado é a integração de TDSGen/OO com a ferramenta JaBUTi/AJ, uma versão da JaBUTi que apóia o teste de programas orientados a aspectos, escritos em AspectJ [7], apenas algumas modificações Módulo Ferramentas de Teste são necessárias. Outra possível extensão do framework está relacionada à satisfação de critérios específicos para o teste de integração.

Agradecimentos

Gostaríamos de agradecer CNPq pelo apoio financeiro.

Referências

[1] Afzal, W. R. Torkar and Feldt, R. A systematic review of search-based testing for non-functional system properties. Information and Software Technology, vol 51 (6), pp 95 –976. June, 2009.

[2] Cheon, Y.; Kim, M. Y and Peruandla, A. 2005. A Complete Automation of Unit Testing for Java Programs. Disponivel em: http://cs.utep.edu/cheon/techreport/tr05-05.pdf.

[3] Ferreira, L. P. and S Vergilio, S. R.. TDSGEN: An Environment Based on Hybrid Genetic Algorithms for Generation of Test Data. In 17th International Conference on Software Engineering and Knowledge Engineering. Taipei, Taiwan, July., 2005

[4] Goldberg, D. E. and Richardson, J. Genetic algorithms with sharing for multimodal function optimization. In Proc of International Conf. on Genetic Algorithms, 1987.

[5] Harman, M. The Current State and Future of Search Based Software Engineering. International Conference on Software Engineering. pp 342-357. 2007

[6] Harrold, M. J. and Rothermel, G. 1994. Performing data flow testing on claases. In: Second ACM SIGSOFT Symposium on Foundations os Software Engineering, New York: ACM Press, 1994, p. 154-163.

[7] Lemos, O.A.L.; Vincenzi, A.M.; Maldonado, J.C. and Masiero, P.C. Control and data flow structural testing criteria for aspect-oriented programs. Journal of Systems and Softwre, v. 80(6), pp 862-882, June, 2007.

[8] Liu, X.; Wang, B.; and Liu, H. Evolutionary search in the context of object-oriented programs. MIC2005: The Sixth Metaheuristics International Conference, Vienna, Austria. 2005

[9] Mantere, T. and Alander, J.T. Evolutionary software engineering, a review. Applied Software Computing. V. 5., pp 315-331, 2005.

[10] McMinn, P., "Search-based software test data generation: a survey," Software Testing, Verification and Reliability, vol. 14, no. 2, pp. 105–156. 2004.

[11] Michael, C.; McGraw, M. C. and Schatz, M.A. Generating Software Test Data by Evolution. IEEE Transactions on Software Engineering, Vol.SE-27(12): 1085-1110. December. 2001.

[12] Muller, R.A. and Holcombe, M. A Symbolic Java virtual machine for test case generation. In: IASTED Conference on Software Engineering, pp 365-371, 2004.

[13] Offut, A. J. and Irvine, A. Testing object-oriented software using the category-partition method. In: XVII International Conference on Technology of Object-Oriented Languages Systems, p. 293 – 304, Santa Barbara, CA, EUA, Agosto. Prentice-Hall. 1995.

[14] Polo, M.; Piattini, M. and Garcia-Rodriguez, I. Decreasing the cost of mutation testing with second-order mutants. Software Testing, Verification & Reliability, v. 19, pp. 111 – 131, 2009.

[15] Ribeiro, J.C.; Veja, F.F. and Zenha-Rela, M.. Using Dynamic Analisis of Java Bytecode for Evolutionary Object-Oriented Unit Testing. Em 25th Brazilian Symposium on Computer Networks and Distributed Systems, Belém/PA. 2007

[16] Ribeiro, J.C. Search-Based Test Case Generation for Object-Oriented Java Software Using Strongly-Typed Genetic Programming., Em Proc. of the GECCO '08, pp. 1819-1822, Atlanta, Georgia, USA, July 2008.

[17] Sagarna, R., A. and Yao, A. X. Estimation of Distribution Algorithms for Testing Object Oriented Software. In: IEEE Congress on Evolutionary Computation (CEC, 2007).

[18] Seesing and H Gross A Genetic Programming Approach to Automated Test Generation for Object-Oriented Software. In: 1st International Workshop on

Evaluation of Novel Approaches to Software Engineering, Erfurt, Germany, September 19—20, 2006.

[19] Tonella, P. Evolutionary Testing of Classes. ISSTA '04: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software testing and Analysis. ACM Press: 119-128. Boston, Massachusetts, USA, 2004.

[20] Vincenzi, M. R; Delamaro, M. E. e Maldonado, J. C. JaBUTi – Java Bytecode Undestanding and Testing. Technical Report –University of São Paulo, March 2003.

[21] Vincenzi, M. R.; .Delamaro, M.E.; Maldonado, J.C. and Wong, E. Establishing Structural Testing Criteria for Java Bytecode. Software Practice and Experience, 36(14): 1.512 – 1.541. Nov. 2006.

[22] Wapller, S. and Lammermann, F. Using evolutionary algorithms for the unit testing of object-oriented software, Proceedings of the 2005 conference on Genetic and evolutionary computation, June 25-29, Washington DC, USA, 2005.

[23] Wappler, S. and Lammermann, F. Evolutionary Unit Testing of Object-Oriented Software Using a Hybrid Evolutionary Algorithm. In: IEEE Congress on Evolutionary Computation, 2006. CEC 2006. Volume , Issue , pp 851 – 858. 2006

[24] Wappler, S. and Wegener, J.: Evolutionary Unit Testing of Object-Oriented Software Using Strongly-Typed Genetic Programming. Em: GECCO '06', Seattle/ Washignton, 2006.

[25] Wegener, J., Baresel, A. and Sthamer, H. Evolutionary test environment for automatic structural testing. Information and Software Technology, 43:841-854, 2001.

Redução do Número de Seqüências no Teste de Conformidade de Protocolos

Jorge Francisco Cutigi, Paulo Henrique Ribeiro, Adenilso da Silva Simão, Simone do Rocio Senger de Souza

[1]Instituto de Ciências Matemáticas e de Computação – Universidade de São Paulo
Caixa Postal 668 – 13.560-970 – São Carlos – SP – Brasil

{jcutigi, phr, adenilso, srocio}@icmc.usp.br

Abstract. *Formal specification is a very important step in the development process of protocol, since the specification can be used as basis to the implementation and the conformance testing. Among formal models for protocol specification, Finite State Machines have been often used. This model allows obtaining test sequences for the specified protocol. Several test generation methods have been developed in past decades, aiming at obtaining a test suite that is able to reveal implementation faults. Nevertheless, most of the generated test suites are huge, with a large number of sequences. In this paper, we present an approach to reduce sequences in a test suite. The main goal is the reduction of the number of sequences, since the large number of sequences can turn the test activity impracticable. We present experimental results, which show that the approach reduces the number of sequences while maintaining the effectiveness in revealing faults.*

Resumo. *A especificação formal é uma etapa crucial no ciclo de desenvolvimento de protocolos, uma vez que ela pode ser usada como base para a implementação e para o teste de conformidade. Dentre os modelos formais de especificação de protocolos, as Máquinas de Estados Finitos têm sido muito utilizadas. Esse modelo permite a derivação de seqüências de teste para o protocolo especificado. Vários métodos de geração de seqüências de teste têm sido desenvolvidos há várias décadas, com o objetivo de obter um conjunto de teste que seja capaz de revelar os defeitos de uma implementação. Entretanto, muitas vezes os conjuntos gerados são muito grandes e possuem um grande número de seqüências. Neste artigo é apresentada uma abordagem de redução de seqüências de teste. Busca-se como objetivo principal a redução do número de seqüências do conjunto de teste, uma vez que o grande número de seqüências pode tornar o teste inviável. São apresentados os resultados de dois estudos experimentais, os quais mostram ganhos consideráveis na redução de seqüências nos conjuntos de teste, mantendo a efetividade em revelar defeitos.*

1. Introdução

Um protocolo é um conjunto de regras que regem a troca de mensagens entre entidades em redes de computadores e em sistemas distribuídos complexos. De acordo com [Sidhu et al. 1991], o desenvolvimento de sistemas baseados em protocolos possui quatro etapas: (1) especificação, que consiste na criação do modelo formal do

protocolo; (2) verificação, que consiste em garantir que a especificação esteja correta; (3) implementação, que é a etapa em que a especificação é transformada em software executável; (4) teste de conformidade, que consiste em confrontar o comportamento da implementação com o comportamento do modelo.

Na fase de especificação, várias técnicas podem ser usadas. Dentre as existentes, as Máquinas de Estados Finitos (MEFs) são muito utilizadas devido a sua simplicidade e capacidade de modelar as propriedades gerais de um protocolo [Bochmann and Petrenko 1994]. Além disso, esse tipo de modelo permite a geração automática de conjuntos de teste por meio de vários métodos. Dentre os métodos mais conhecidos, pode-se destacar o método W [Chow 1978], Wp [Fujiwara et al. 1991], HSI [Petrenko et al. 1993, Luo et al. 1994] e H [Dorofeeva et al. 2005].

Grande parte dos métodos existentes geram testes compostos de várias seqüências distintas que devem ser aplicadas no estado inicial do sistema. Em geral, assume-se a existência de um operação *reset*, que leva tanto a MEF quanto sua implementação ao seu estado inicial. O *reset* deve ser inserido no início de cada seqüência do conjunto de teste, portanto, o número de operações *resets* é igual ao número de seqüências de um conjunto de teste. Essa operação pode ainda aumentar o custo do teste [Hierons 2004, Yao et al. 1993, Fujiwara et al. 1991]. Além disso, deve-se assumir que a operação *reset* está implementada de maneira correta [Fujiwara et al. 1991]. Sendo assim, é desejável que um conjunto de teste tenha o mínimo de operações *reset* possível [Hierons 2004, Hierons and Ural 2006].

Como uma solução para o problema do uso de operações *reset*, alguns métodos de geração de conjuntos de teste geram seqüências de verificação [Hennie 1964, Gonenc 1970, Ural et al. 1997, Hierons and Ural 2002, Chen et al. 2005, Ural and Zhang 2006, Hierons and Ural 2006, Simao and Petrenko 2008], que se trata da geração de um conjunto de teste unitário, ou seja, com apenas uma seqüência de teste. Nesses casos, considerados como ideais, o número de seqüências e, conseqüentemente, o número de *resets*, correspondem ao mínimo possível. Porém, esses métodos requerem que a MEF possua uma seqüência de distinção, contudo nem todas as MEFs minimais possuem seqüências de distinção [Lee and Yannakakis 1994].

Os métodos de geração de conjuntos de teste citados têm a propriedade de gerarem conjuntos completos. Um conjunto de seqüências de teste é chamado de *completo* se ele é capaz de detectar todos os defeitos em um determinado domínio. Geralmente, o domínio de defeitos é definido em função do número máximo de estados que uma máquina de estados deve ter para ser equivalente à implementação. Nos conjuntos de teste completos, as seqüências podem ser combinadas para se reduzir o número de seqüências. No entanto, essa abordagem pode reduzir a efetividade do conjunto em revelar defeitos. Dessa forma, a combinação de seqüências, visando a redução do número de seqüências, deve idealmente preservar a completude do conjunto.

Um tópico que possui estreita relação com os conjuntos de teste e que está presente implícita ou explicitamente nos diversos métodos encontrados na literatura

são as condições de suficiência para completude de casos de teste. Um conjunto de condições de suficiência determina quais são as condições que tornam um conjunto de testes completo. Os diversos métodos de geração garantem que o conjunto gerado satisfaz algum conjunto de condições de suficiência e, portanto, possui a propriedade de ser completo. Entretanto, para oferecer essa garantia, normalmente são gerados conjuntos maiores que o necessário. Sendo assim, é possível a utilização de condições de suficiência na geração, minimização e redução de conjuntos de seqüências de teste.

Neste artigo é apresentado um algoritmo para redução do número de seqüências de um conjunto de casos de teste para especificações de protocolos em MEFs, preservando a completude do conjunto. A abordagem se baseia na combinação de seqüências de um conjunto completo. Com essa combinação, o número de seqüências do conjunto é reduzido, além de que a seqüência gerada pela combinação pode gerar redundâncias, o que torna possível uma diminuição no tamanho dessa seqüência. Para garantir a completude do conjunto obtido, verifica-se se as seqüências do novo conjunto satisfazem certas condições de suficiência. Os resultados de uma avaliação experimental são apresentados, avaliando a efetividade da proposta em relação à porcentagem de redução do número de *resets*. Os resultados mostram uma redução de até 80% no número de operações *resets* em relação aos métodos clássicos de geração.

Este artigo está organizado da seguinte forma: na Seção 2 são abordados os principais conceitos relacionados a MEFs, assim como as definições necessárias para o entendimento deste artigo. Na Seção 3 é apresentada a abordagem de redução de *resets*, com os detalhes do algoritmo, análises e um exemplo. Na Seção 4 são apresentados os resultados de uma avaliação experimental da abordagem de redução de *resets*, contendo um estudo de caso em um protocolo de comunicação. Por fim, na Seção 5 são apresentadas as conclusões do artigo, com as limitações da abordagem e trabalhos futuros.

2. Máquina de Estados Finitos

Segundo [Gill 1962], uma MEF é uma máquina hipotética composta por estados e transições, definida a seguir.

Definição 1 *Uma MEF M (determinística e de* Mealy*) é uma tupla $(S, s_1, X, Y, D, \delta, \lambda)$, onde:*

- *S é um conjunto finito de estados, incluindo o estado inicial s_1.*
- *X é um conjunto finito de entradas.*
- *Y é o conjunto finito de saídas.*
- *D é o domínio da especificação, $D \subseteq S \times X$*
- *δ é uma função de transição, $\delta : D \rightarrow S$.*
- *λ é uma função de saída, $\lambda : D \rightarrow Y$.*

Se $D \rightarrow S \times X$ então a MEF é completamente especificada ou completa. Caso contrário, ela é chamada de parcialmente especificada ou parcial. Uma tupla $(s, x) \in D$ é uma transição definida de M. Um seqüência $\alpha = x_1 x_2 ... x_k$ é dito ser uma seqüência de entrada definida no estado $s \in S$ se existe $s_1, s_2, ..., s_{k+1}$, onde $s_1 = s$, tal que $(s_i, x_i) \in D$ e $\delta(s_i, x_i) = s_{i+1}$ para todo $1 < i < k$. Denota-se por $\Omega(s)$ o conjunto de todas as seqüências de entradas definidas no estado s.

Uma MEF pode ser representada por um grafo direcionado, no qual cada estado é representado por um vértice e cada transição é representada por uma aresta direcionada. O estado inicial é indicado por uma seta incidente ao nó da MEF. Cada aresta possui um rótulo que indica o par *entrada/saída* e o próximo estado. Um exemplo dessa representação é apresentado na Figura 1. O conjunto S de estados representa o conjunto de todas as configurações possíveis do sistema em relação aos símbolos de entrada e saída. A MEF da Figura 1 possui o conjunto de estados $S = \{s_1, s_2, s_3, s_4\}$, o alfabeto de entrada $X = \{a, b\}$ e o alfabeto de saída $Y = \{0, 1\}$.

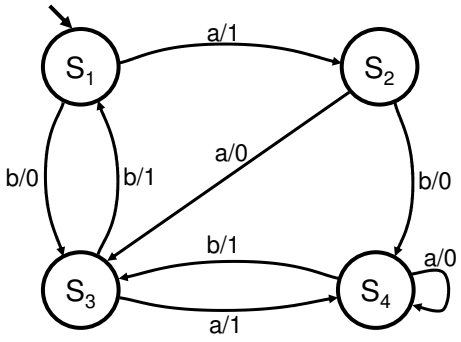


Figura 1. MEF

Uma MEF é minimal se nenhum par de estados da máquina é equivalente, ou seja, para todo par de estados (s_i, s_j), existe ao menos uma seqüência de entradas, definidas em s_i e s_j, cujas saídas produzidas sejam distintas. Uma MEF é fortemente conexa se para todo par de estados (s_i, s_j), existe uma seqüência de entradas, definidas em s_i, que leve de s_i até s_j.

A função de transição δ e a função de saída λ são estendidas para seqüência de entradas, incluindo a seqüência vazia, que é denotada por ϵ. Tem-se que $\delta(s, \epsilon) = s$ e $\lambda(s, \epsilon) = \epsilon$ para todo $s \in S$. Seja β uma seqüência de entradas e $\delta(s, \beta) = s'$, então, para todo $x \in X$ define-se $\delta(s, \beta x) = \delta(s', x)$ e $\lambda(s, \beta x) = \lambda(s, \beta)\lambda(s', x)$. Uma seqüência de transferência χ de s_i para s_j, é uma seqüência que conduz M do estado s_i para o estado s_j, ou seja, $\delta(s_i, \chi) = s_j$.

Dois estados $s_i, s_j \in S$ são equivalentes se existe $\gamma \in \Omega(s_i) \cap \Omega(s_j)$, tal que $\lambda(s_i, \gamma) = \lambda(s_j, \gamma)$. Esse conceito pode ser aplicado em estados de MEFs diferentes. MEFs são equivalentes se seus estados iniciais são equivalentes. De forma análoga, dois estados, $s_i, s_j \in S$ são distinguíveis se existe uma seqüência $\gamma \in \Omega(s_i) \cap \Omega(s_j)$ tal que $\lambda(s_i, \gamma) \neq \lambda(s_j, \gamma)$. MEFs são distinguíveis se seus estados iniciais são distinguíveis.

A operação *reset* é uma operação especial que leva a MEF de qualquer estado para o estado inicial e com saída nula. É denotada pela letra r e aparece como primeiro símbolo em uma seqüência de teste. O tamanho de um conjunto de seqüências de teste é obtido pelo número de símbolos de entrada contido no conjunto adicionado com o número de operações *resets*.

Na geração de testes a partir de MEFs, assume-se que a implementação pode ser modelada como uma MEF contida em um domínio de defeitos. Essa hipótese, conhecida como hipótese de teste, é necessária para que um conjunto fi-

nito de testes possa ser gerado [Chow 1978, Ural et al. 1997, Hierons and Ural 2006, Hennie 1964]. $\Im(M)$ denota o domínio de defeitos definido pelo conjunto de todas as MEFs com o mesmo alfabeto de entrada e no máximo o mesmo número de estados de M, utilizado por grande parte dos métodos de geração, como por exemplo, os métodos W [Chow 1978], Wp [Fujiwara et al. 1991], HSI [Petrenko et al. 1993, Luo et al. 1994], H [Dorofeeva et al. 2005], entre outros.

Um caso de teste T é *completo* se para cada MEF $N \in \Im(M)$ tal que N e M são distinguíveis, existe uma seqüência pertencente a T que distingue N de M. Ou seja, se o caso de teste é completo, ele é capaz de revelar todos os defeitos de uma implementação de M que possa ser modelada por uma MEF de $\Im(M)$.

Condições de suficiência determinam quais são as propriedades que tornam um caso de teste capaz de revelar todos os defeitos de um dado domínio. Ou seja, se determinado caso de teste satisfaz as condições de suficiência, garante-se que esse caso de teste é completo. Condições de suficiência para conjuntos de seqüências de teste foram definidas em [Petrenko et al. 1996], as quais permitiram provar a completude de diversos métodos de geração já existentes. Em [Dorofeeva et al. 2005] foram definidas novas condições que generalizavam as condições de [Petrenko et al. 1996]. Posteriormente, em [Simao and Petrenko 2009] foram definidas condições de suficiência mais abrangentes, as quais generalizam todas as anteriores.

Um algoritmo para a verificação da completude de casos de teste também é apresentado em [Simao and Petrenko 2009], o qual foi utilizado para verificar a completude dos conjuntos no presente trabalho.

3. Redução de *resets*

Os conjuntos de teste gerados pelos métodos clássicos geralmente apresentam redundâncias, ou seja, há seqüências ou subseqüências que poderiam ser descartadas ou substituídas por outra de tamanho menor. Com isso, obtém-se um conjunto menor, mantendo a propriedade de completude do conjunto. Uma estratégia que também pode diminuir o conjunto é a combinação de seqüências. Essa estratégia consiste na concatenação das seqüências de teste de forma a manter a completude do conjunto. A redução do número de seqüências é relevante no sentido de que o número de operações *reset* também diminui.

Neste artigo, é proposta uma estratégia de redução do número de seqüências de conjuntos de seqüências de teste. A estratégia consiste em dois passos: (1) Concatenar as seqüências de um conjunto de seqüências de teste, de forma a obter um conjunto ainda completo e com menos seqüências que o conjunto original; (2) A partir do conjunto gerado no passo 1, identificar em cada seqüência do conjunto concatenado o menor prefixo dela que ainda mantém a completude do conjunto, para que assim sejam eliminados símbolos de entrada desnecessários.

O processo de concatenação de seqüências utilizado no passo 1 do algoritmo é realizado de duas formas:

- **Sobreposição:** Essa forma de concatenação consiste na sobreposição de entradas em duas seqüências de entrada. A concatenação por sobreposição

se dá em uma seqüência $\alpha = \chi\phi$ concatenada com uma seqüência $\beta = \phi\gamma$, gerando uma seqüência $\omega = \alpha\gamma$, onde $\delta(s_1, \chi) = s_1$. Neste artigo, a operação de concatenação por sobreposição é denotada por $concatSP(\alpha, \beta)$.

- **Seqüência de transferência:** Ao fim da seqüência α deve ser adicionada uma seqüência de transferência χ que leva a MEF do estado $s = \delta(s_1, \alpha)$ ao estado inicial s_1. Antes de concatenar a seqüência β escolhe-se a menor seqüência de transferência. Dessa forma, a concatenação final resulta em $\omega = \alpha\chi\beta$. Neste artigo, a operação de concatenação por seqüência de transferência é denotada por $concatST(\alpha, \beta)$.

A concatenação por sobreposição é preferível em relação à que utiliza seqüências de transferência, uma vez que, com a sobreposição, entradas são reaproveitadas, o que leva a uma diminuição no tamanho da seqüência ω. O contrário acontece com o uso de seqüência de transferência, em que o tamanho da seqüência ω obtida é maior que a soma dos tamanhos de α e β, o que aumenta o tamanho da seqüência. Nota-se que essas duas formas de concatenação mantém em ω as transições originais de α e β. A Figura 2 ilustra as duas estratégias de concatenação, em que a seqüência $\alpha = aabab$ é concatenada por sobreposição com a seqüência $\beta = abaa$, que resulta na seqüência $aababaa$. A seqüência $\beta = abaa$ também é concatenada com a seqüência $\gamma = aaab$ por meio da seqüência de transferência $\chi = bb$, resultando na seqüência $abaabbaaab$.

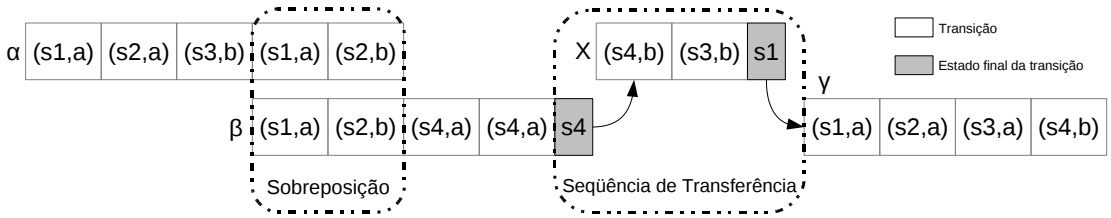


Figura 2. Sobreposição e Seqüência de Transferência

O algoritmo de redução recebe como entrada uma MEF determinística, minimal, de *Mealy* e fortemente conexa, e um conjunto de seqüências de teste. O algoritmo é dividido em dois passos, como seguem.

O **passo 1** do algoritmo tem como objetivo a obtenção de um conjunto com menos seqüências e que ainda seja completo. Tem-se um conjunto completo TS gerado por algum método de geração. Com isso, seleciona-se duas seqüências α e β, tal que $\{\alpha, \beta\} \subseteq TS$, que serão concatenadas de acordo com o processo de concatenação citado, gerando uma seqüência ω definida no estado inicial. Feito isso, é verificado se o conjunto $TS - \{\alpha\} - \{\beta\} \cup \{\omega\}$ satisfaz as condições de suficiência, isto é, que ele ainda é completo. Caso positivo, remove-se α e β de TS, insere ω e repete-se o processo com TS. Caso contrário, repete-se o processo com outra escolha de seqüências α e β. Ao fim, tem-se o conjunto TS completo e com um número menor de seqüências. Em resumo, o passo 1 recebe o conjunto TS e o concatena o máximo de seqüências possíveis utilizando a estratégia de sobreposição. Com o conjunto gerado, a forma de concatenação por seqüência de transferência pode ser utilizada. O algoritmo 1 apresenta o passo 1.

Data: MEF M e Conjunto TS de seqüências de teste
Result: Conjunto TS concatenado
while *existe* $\alpha, \beta \in TS$, *tal que* $TS - \{\alpha, \beta\} \cup \{concatSP(\alpha, \beta)\}$ *é completo* **do**
    $TS \leftarrow TS - \{\alpha, \beta\} \cup \{concatSP(\alpha, \beta)\}$;
end
while *existe* $\alpha, \beta \in TS$, *tal que* $TS - \{\alpha, \beta\} \cup \{concatST(\alpha, \beta)\}$ *é completo* **do**
    $TS \leftarrow TS - \{\alpha, \beta\} \cup \{concatST(\alpha, \beta)\}$;
end
return TS

Algoritmo 1: Algoritmo do passo 1

Exemplificando o processo de concatenação descrito, considere a MEF da Figura 1 e o conjunto completo gerado pelo método W

$$TS = \{raabb, rbabb, raaabb, rababb, rbaabb, rbbabb, rabaabb, rabbabb\}$$

de tamanho 48 e 8 operações *resets*.

Toma-se $\alpha = aabb$ e $\beta = babb$, ambas sem as operações *resets*. Nesse caso, a concatenação por sobreposição é possível, pois a seqüência α pode ser decomposta em $\chi = aab$ e $\phi = b$, com $\delta(s_1, \chi) = s_1$, e a seqüência β pode ser decomposta em $\phi = b$ e $\gamma = abb$. Com isso, tem-se $\omega = concatSP(\alpha, \beta) = \alpha\gamma = aabbabb$. Obtém-se o conjunto $TS = \{raabbabb, raaabb, rababb, rbaabb, rbbabb, rabaabb, rabbabb\}$. Verifica-se que esse conjunto satisfaz as condições de suficiência propostas em [Simao and Petrenko 2009]. A partir desse conjunto resultante, repete-se o mesmo procedimento. Neste exemplo, apenas mais uma sobreposição é possível, com $\alpha = bbabb$ e $\beta = abbabb$), que resulta em $\omega = bbabbabb$. Com isso, tem-se o conjunto completo $TS = \{raabbabb, rbbabbabb, raaabb, rababb, rbaabb, rabaabb\}$.

Após não haver mais possibilidade do uso de sobreposição na concatenação, faz-se então o uso de seqüências de transferência. Exemplificando, toma-se $\alpha = aabbabb$ e $\beta = aaabb$. Nesse caso, uma seqüência de transferência χ deve ser inserida entre α e β. Tem-se então $\delta(s_1, aabbabb) = s_1$, o que indica que χ deve ser uma seqüência que leva a MEF do estado s_1 ao próprio estado inicial s_1, resultando na seqüência vazia $\chi = \epsilon$. Com isso, tem-se a seqüência $\omega = \alpha\chi\beta$, que resulta na seqüência $\omega = aabbabbaaabb$. Com isso tem-se o conjunto $TS = \{raabbabbaaabb, rbbabbabb, rababb, rbaabb, rabaabb\}$. Verifica-se que o conjunto satisfaz as condições de suficiência propostas em [Simao and Petrenko 2009]. Ao fim do processo, tem-se o conjunto completo resultante $TS = \{raabbabbaaabbbbabbabbababbbaabbabaabb\}$. Nota-se que o conjunto agora contém apenas uma seqüência, o que indica a redução máxima de *resets* que pode ser obtida.

Após a criação do conjunto TS concatenado, tem-se o **passo 2** do algoritmo, o qual consiste na redução desse conjunto TS. Nessa etapa, para cada seqüência α em TS, deve-se identificar o menor prefixo β de α necessário para ainda manter a completude do conjunto. Com isso, remove-se símbolos de entradas desnecessários. O passo 2 é apresentado no algoritmo 2.

```
Data: MEF M e Conjunto TS de seqüências de teste
Result: Conjunto TS_reduzido
1 for cada seqüência α ∈ TS do
2     seja β o menor prefixo de α tal que TS − {α} ∪ {β} é completo;
3     TS ← TS − {α} ∪ {β};
4 end
5 return TS;
```

Algoritmo 2: Algoritmo do passo 2

Por exemplo, tomando-se o conjunto $TS = \{raabbabbaaabbbbabbabbababbbaabbabaabb\}$, tem-se apenas uma seqüência, a qual apenas o prefixo $\beta = aabbabbaaabbbbabbabbababbbaabb$ é necessário para manter a completude do conjunto.

Ao fim da execução dos passos 1 e 2, o conjunto final e reduzido da abordagem é

$$TS \text{ reduzido} = \{raabbabbaaabbbbabbabbababbbaabb\}$$

de tamanho 31 e com 1 operação *reset*, o que indica uma redução de 87,5% no número de operações *resets* e uma redução de 35,4% em relação ao tamanho do conjunto.

Diferentes resultados podem ser obtidos com a execução de uma mesma MEF e um mesmo conjunto de seqüências, dependendo da ordem em que as seqüências são consideradas. Para evitar que o algoritmo tenha o desempenho influenciado pela ordem das seqüências, elas são selecionadas de forma aleatória. Executando-se 10 vezes o algoritmo com o exemplo apresentado, a média de redução do tamanho do conjunto foi de de 47,5% e a média de redução de *resets* foi de 77,6%, demonstrando que o resultado do exemplo apresentado é significativo. Deve-se ressaltar que o conjunto de testes obtidos possui o mesmo poder de revelar defeitos modelados pelo domínio de defeitos.

4. Avaliação Experimental

A fim de avaliar a abordagem proposta, estudos experimentais foram conduzidos de forma a verificar a redução do número de operações *resets* no conjunto final. Para isso, dois estudos foram realizados. O primeiro trata de experimentos conduzidos com MEFs aleatórias. O segundo estudo aplica a redução de *resets* em um protocolo de comunicação.

No primeiro estudo experimental foram geradas de maneira aleatória MEFs completas e minimais com 5 entradas, 5 saídas e com o número n de estados variando de 4 a 15, sendo que para cada valor de n foram geradas 30 MEFs diferentes, totalizando 360 MEFs. O processo de geração das MEFs, descrito em [Simao and Petrenko 2009] é realizado em três etapas: Na primeira etapa, um estado é selecionado com estado inicial e marcado como alcancável. Então, para cada estado estado s não marcado como alcançável, o gerador aleatório seleciona um estado alcançável s', uma entrada x e uma saída y e adiciona uma transição de s' para s com entrada x e saída y, marcando s como alcançável. Feito isso, tem-se início a segunda etapa, em que o gerador adiciona, se necessário, mais transições aleatórias.

Na terceira etapa, é verificado se a MEF é minimal. Caso não seja minimal, a MEF é descartada e uma outra MEF é gerada.

Os conjuntos gerados para realizar a redução foram obtidos a partir dos métodos W [Chow 1978], HSI [Petrenko et al. 1993, Luo et al. 1994] e H [Dorofeeva et al. 2005]. Portanto, foram gerados três conjuntos de seqüências para cada uma das 360 MEFs, totalizando 1080 conjuntos de seqüências de teste.

Considerando os conjuntos obtidos pelo método W, os resultados obtidos após a condução dos experimentos evidenciaram uma redução média de *resets* em 89,4% com desvio padrão de 8,1%. Em relação aos conjuntos gerados pelo método HSI, os *resets* foram reduzidos em 82,3%, com desvio padrão de 14,8%. Já em relação aos conjuntos gerados pelo método H, os resultados mostraram uma redução média de 71,2% com desvio padrão de 16,3%. Considerando os três métodos, a média de redução de *resets* foi de 80,9%. Esses dados são apresentados na Tabela 1.

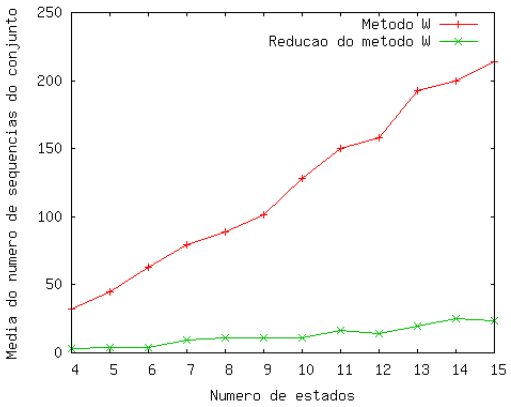
Tabela 1. Resultados gerais para o Estudo 1

Método	Redução de *resets* obtida	Desvio Padrão
W	89,4%	8,1%
HSI	82,3%	14,8%
H	71,2%	16,3%
Média	80,9%	

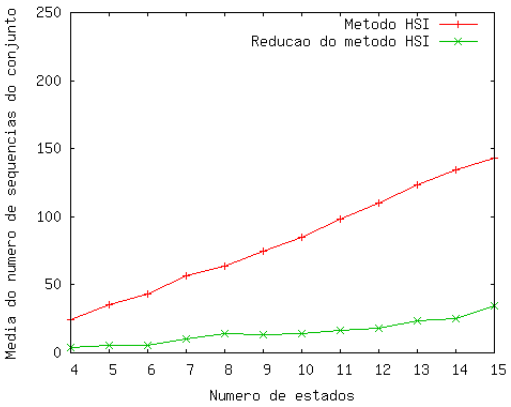
Os gráficos da Figura 3.a, 3.b e 3.c mostram o crescimento médio do número de *resets* pelos métodos W, HSI e H, respectivamente. Nota-se que o crescimento de *resets* obtido na redução é bem menor que o crescimento obtido na geração. Os gráficos mostram também que as taxas de crescimento de *resets* da redução dos métodos H e HSI são muito parecidos, apesar do método HSI gerar conjuntos maiores que o método H. Outro fato interesante inferido por meio dos gráficos é que os números de seqüências do conjunto resultante da redução são bem parecidos, independente do método utilizado.

A Figura 3.d apresenta um gráfico boxplot [McGill et al. 1978], que representa a distribuição dos dados das taxas de redução para cada método. Nos dados do método H, a taxa de redução se concentra na maior parte dos casos entre 30% e 98%, com maior densidade entre 60% e 85% e um *outlier* inferior que aponta um caso em que a redução não foi possível. Para o método HSI, os dados mostram uma taxa de redução entre 50% e 98%, com maior densidade entre 70% e 90%. Porém, a redução do método HSI apresenta alguns *outliers* inferiores, que indicam que em alguns casos a redução foge da área de maior densidade apresentada no gráfico. O comportamento em relação ao método W é parecido com o HSI, em que *outliers* também aparecem. Porém, o intervalo de maior densidade dos dados é menor que os método H e HSI, ou seja, as taxas de redução são muito próximas, independente da MEF e do conjunto W reduzido.

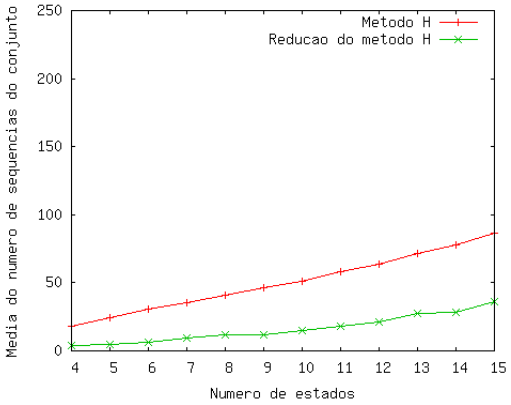
O segundo estudo experimental foi baseado na aplicação da abordagem de redução envolvendo uma especificação em MEF de um protocolo. O estudo de caso é construído sobre o protocolo INRES (INitiator-RESponder) [Tan et al. 1996,



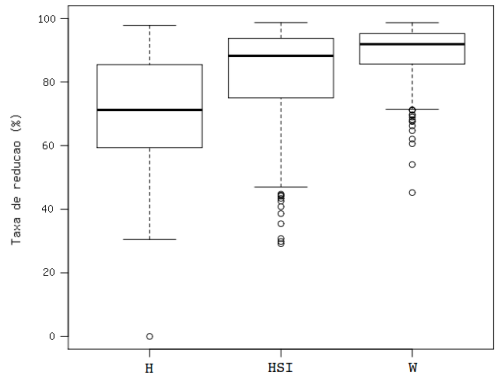
(a) Variação do número de seqüências para o método W



(b) Variação do número de seqüências para o método HSI



(c) Variação do número de seqüências para o método H



(d) Boxplot das taxas de redução de cada método

Figura 3. Gráficos do primeiro estudo experimental

Hogrefe 1991], que contém aspectos essencias dos protocolos de comunicação e mantém a regra de comunicação entre as duas entidades do protocolo: *Initiator* e *Responder*. Na Figura 4 é apresentada a MEF que especifica o comportamento do *Responder* do protocolo INRES. A MEF é determinística, reduzida e parcialmente especificada, a qual contém 4 estados, 5 entradas, 7 saídas e 11 transições.

Por se tratar de uma MEF parcial, os métodos HSI e H foram utilizados para a geração do conjunto de seqüências de teste, os quais foram submetidos ao processo de redução das operações *resets*. Cada conjunto de seqüências de teste foi submetido 10 vezes no processo de redução, de modo a obter um resultado sem influências da escolha aleatória de seqüências. O conjunto gerado pelo método HSI contém 21 operações *resets* e, quando aplicada a redução, esse número é reduzido em 65%. O conjunto gerado pelo método H contém possui 17 *resets*, o qual foi reduzido a uma taxa de 62%. Deve-se observar que, por se tratar de uma MEF parcial, o número de seqüências geradas é geralmente menor e, conseqüentemente, as possibilidades de redução são menores. Contudo, obteve-se uma redução significativa.

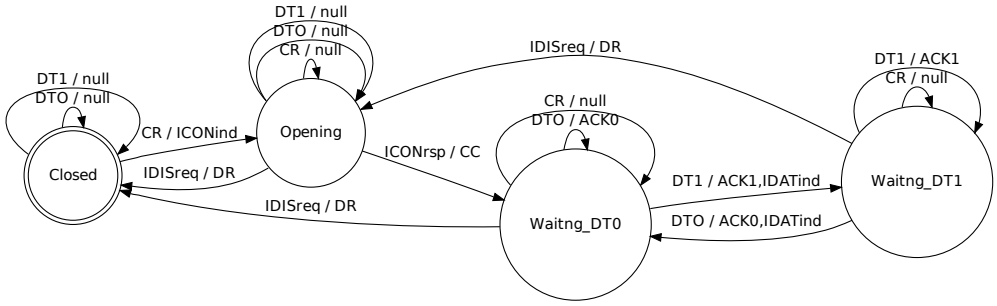


Figura 4. Protocolo INRES – Responder

5. Conclusões

Neste artigo foi investigado o problema da redução de conjuntos de seqüências de teste a partir de MEFs. Em particular, o problema da redução do número de operações *resets* foi priorizado, uma vez que essa operação, em grande parte das vezes, é muito custosa para a aplicação. Para isso, um algoritmo de redução baseado nas condições de suficiência [Simao and Petrenko 2009] foi proposto. A abordagem mostrou ganhos significativos, os quais foram comprovados por meio da condução de uma avaliação experimental com conjuntos gerados pelos métods W, HSI e H. Ganhos médios de 80% na redução de operações *resets* foram verificados. Com isso, mostrou-se também a viabilidade das condições de suficiência definidas em [Simao and Petrenko 2009], que são capazes de reconhecer conjunto menores que os gerados pelos métodos citados.

Como perspectivas para trabalhos futuros, o problema da concatenação aleatória de seqüências é um ponto em que pode haver evoluções. A identificação de propriedades nas seqüências podem direcionar uma melhor escolha delas, otimizando o processo final de redução, além de escolher duas seqüências que obrigatoriamente gerariam ainda um conjunto completo.

Referências

Bochmann, G. V. and Petrenko, A. (1994). Protocol testing: review of methods and relevance for software testing. In *ISSTA '94: Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, pages 109–124, New York, NY, USA. ACM.

Chen, J., Hierons, R. M., Ural, H., and Yenigun, H. (2005). Eliminating redundant tests in a checking sequence. In *TestCom 2005*, number 3502 in lncs, pages 146–158.

Chow, T. S. (1978). Testing software design modeled by finite-state-machines. *IEEE Transactions on Software Engineering*, 4(3):178–186.

Dorofeeva, R., El-Fakih, K., and Yevtushenko, N. (2005). An improved conformance testing method. In *FORTE*, pages 204–218.

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

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

Gonenc, G. (1970). A method for design of fault detection experiments. *IEEE Transactions on Computers*, 19(6):551–558.

Hennie, F. C. (1964). Fault detecting experiments for sequential circuits. pages 95–110.

Hierons, R. M. (2004). Using a minimal number of resets when testing from a finite state machine. *Inf. Process. Lett.*, 90(6):287–292.

Hierons, R. M. and Ural, H. (2002). Reduced length checking sequences. *IEEE Transactions on Computers*, 51(9):1111–1117.

Hierons, R. M. and Ural, H. (2006). Optimizing the length of checking sequences. *IEEE Transactions on Computers*, 55(5):618–629.

Hogrefe, D. (1991). Osi formal speci cation case study: the inres protocol and service. Technical report, University of Bern.

Lee, D. and Yannakakis, M. (1994). Testing finite-state machines: State identification and verification. *IEEE Trans. Comput.*, 43(3):306–320.

Luo, G., Petrenko, R., and Bochmann, G. V. (1994). Selecting test sequences for partially-specified nondeterministic finite state machines. In *In IFIP 7th International Workshop on Protocol Test Systems*, pages 91–106.

McGill, R., Tukey, J. W., and Larsen, W. A. (1978). Variations of box plots. *The American Statistician*, 32(1):12–16.

Petrenko, A., von Bochmann, G., and Yao, M. Y. (1996). On fault coverage of tests for finite state specifications. *Computer Networks and ISDN Systems*, 29(1):81–106.

Petrenko, A., Yevtushenko, N., Lebedev, A., and Das, A. (1993). Nondeterministic state machines in protocol conformance testing. In *Protocol Test Systems*, pages 363–378.

Sidhu, D., Chung, A., and Blumer, T. P. (1991). Experience with formal methods in protocol development. *SIGCOMM Comput. Commun. Rev.*, 21(2):81–101.

Simao, A. S. and Petrenko, A. (2008). Generating checking sequences for partial reduced finite state machines. In *TestCom '08 / FATES '08: Proceedings of the 20th IFIP TC 6/WG 6.1 international conference on Testing of Software and Communicating Systems*, pages 153–168, Berlin, Heidelberg. Springer-Verlag.

Simao, A. S. and Petrenko, A. (2009). Checking fsm test completeness based on sufficient conditions. *IEEE Transactions on Computers*. (Aceito para publicacao. Versão preliminar disponível em: www.crim.ca/Publications/2007/documents/plein_texte/ASD_PetA_0710_20.pdf).

Tan, Q. M., Petrenko, A., and Bochmann, G. V. (1996). A test generation tool for specifications in the form of state machines. In *in Proceedings of the International Communications Conference*, pages 225–229.

Ural, H., Wu, X., and Zhang, F. (1997). On minimizing the lengths of checking sequences. *IEEE Transactions on Computers*, 46(1):93–99.

Ural, H. and Zhang, F. (2006). Reducing the lengths of checking sequences by overlapping. *Lecture Notes on Computer Science*, (3964):274–288.

Yao, M., Petrenko, A., and Bochmann, G. v. (1993). Conformance testing of protocol machines without reset. In *Proceedings of the IFIP TC6/WG6.1 Thirteenth International Symposium on Protocol Specification, Testing and Verification XIII*, pages 241–256, Amsterdam, The Netherlands, The Netherlands. North-Holland Publishing Co.

Embedded Critical Software Testing for Aerospace Applications based on PUS

Rodrigo P. Pontes[1], Eliane Martins[2], Ana M. Ambrósio[3], Emília Villani[1]

[1]Instituto Tecnológico de Aeronáutica (ITA)
Vila das Acácias, CEP 12.228-900 – São José dos Campos – SP – Brazil

[2]Instituto de Computação – Universidade Estadual de Campinas (Unicamp)
Avenida Albert Einstein, 1251, CEP 13083-970, Campinas – SP – Brazil

[3]Departamento de Sistemas e Solo – Instituto Nacional de Pesquisas Espaciais (INPE)
Avenida dos Astronautas, 1758, CEP 12227-010, São José dos Campos – SP – Brazil

{rpastl,evillani}@ita.br, eliane@ic.unicamp.br, ana@dss.inpe.br

Abstract. *This paper discusses the practical experience of verifying an On-Board Data Handling (OBDH) software to be used in a future satellite application at INPE using the CoFI testing methodology. This technique is proper for aerospace applications and is based on modeling the system under test as finite state machines. The test cases are automatically generated from the developed models. The OBDH software considered in this paper follows the PUS standard from European Cooperation for Space Standardization, which is being adopted in Brazil. Among the important issues analyzed by this practical experience are the errors found, the time required for the modeling activity, the time required for testing, the reusability of the test cases, among others.*

1. Introduction

During last decades, the software role in space embedded systems has increased. However, the attention and efforts dedicated to its design and verification have not increased in the same way. Hardware is still the main concern of the development of embedded systems. When the project resources are limited, the efforts are addressed to hardware issues rather than software. As a consequence, software is also playing a significant role in accidents, Leveson (2005).

Considering this scenario, this work analyzes one specific technique for the verification of space embedded software: the CoFI (*Conformance and Fault Injection*), Ambrosio (2005). CoFI is a model based testing methodology that uses state machines to represent the behavior of the system under different assumptions. The test cases are generated automatically from these models and they are applied to the system under test.

The main purpose of this work is not to compare this methodology to others, but to identify the advantages and the limits of its utilization through a practical experience, a practical case study. The comparison among others testing methodologies were performed in Ambrosio(2005).

This work discusses the results of the application of the CoFI testing methodology into a case study in the space area: the on-board data handling (OBDH) software to be used in a future satellite application at INPE. This OBDH is being developed using an object-oriented implementation, Arias et al. (2008).

The OBDH software is based on the PUS (Package Utilization Standard), a proposal of the European Cooperation on Space Standardization (ECSS) that have also been adopted in Brazil. The use of standards in space area has been motivated by time-saving and dependability-improvement of the software development. The application of a testing methodology based on models that are derived from a standard will consequently reduce the cost with tests. The modeling process performed in this work is general, because it is developed from the PUS standard.

For this case study, important issues related to the CoFI applicability are discussed, such as the size of the models, the time spent on modeling the system, the time spent on the application of the tests, the number of errors detected, how critical the detected errors are, among others.

This work is organized as follows: Section 2 introduces the CoFI methodology and the Condado tool. Section 3 discusses previous works developed with the CoFI. Section 4 presents the PUS standard and details the telecommand verification service. This service is used is Section 5 to present the application of CoFI to the OBDH software including the models, the test cases and the results. The Section 6 brings some conclusions and discusses the contributions of this work.

2. CoFI Testing Methodology and the Condado Tool

The CoFI Testing Methodology consists of a systematic way to create test cases for space software. The CoFI is comprised of steps to identify a set of services. Each service is represented in finite state machines. The models represent the behavior of the System Under Test (SUT) under the following *classes of inputs* arriving: (i) normal, (ii) specified exceptions, (iii) inopportune inputs and (iv) invalid inputs caused by hardware faults.

The software behavior is represented by small models taking into account the decomposition in terms of: (i) the services provided by software and (ii) the types of behavior under the classes of inputs. The *types of behavior* defined in the context of the CoFI are: Normal, Specified Exception, Sneak Path, and Fault Tolerance. These behaviors are respectively associated to the following inputs: normal, specified exceptions, inopportune and invalid inputs. More than one model can be created in order to represent a type of behavior for a given service.

After the creation of the partial models, each model is submitted to the Condado tool that is able to tour the model, Martins (1999). The Condado tool generates the test cases from these models combining different sequences of inputs. Each test case is a sequence of inputs and its expected output(s) associated to the transitions of a tour. Each tour ends in the final state, if the final state is the initial state, the application of the test cases is performed without restarting the system. The CoFI test case set is the union of the test cases generated from each model and must cover all the transitions of the finite state machine models.

The main reasons of choosing the Condado tool are: the generation of independent test cases; the cover of all transitions of the model; availability of the tool; and the validity of all test cases. The latter is justified by the fact that all test cases are a sequence of inputs that starts in an initial state and they are led to a final state. The disadvantage is the generation of possible repeated test cases.

The generation of the test cases manually, besides being a tough task, might introduce errors during the process. Thus, as there was already an automatic tool for generating test cases, it was used.

It is not crucial the utilization of the Condado tool. Other tools can be used, provided that it covers, at least, all the transitions of the model and it accepts partial finite state machines.

3. Other CoFI Applications

In Ambrosio et al. (2008), the CoFI testing methodology was applied in the context of an independent software verification and validation process of the Quality Software Embedded in Space Missions (QSEE) Project carried on at INPE. The software under test in the QSEE Project is the software embedded in the Payload Data Handling Computer, which is part of a scientific X-ray instrument onboard of a scientific satellite under development at INPE. The CoFI methodology served as a guideline to focus the tester's attention on the faults and exceptions that occur during the software's operation, leading to situations that the developers had not thought of.

Pontes et al. (2009) compares two different verification techniques: model checking and the CoFI Test methodology. It uses an automatic coffee machine example as a case study to show the contributions of each technique. Because of weak points identified in both techniques, the work conclusion is that the two techniques are complementary to each other. The main contributions of the techniques are the detection of incomplete and inconsistent requirements, the introduction of testability requirements and an adequate treatment of all exceptions.

In Morais and Ambrosio (2010), an adaptation of the CoFI is proposed to be applied in the initial phases of the software development, as a new approach to refine software requirements. The new approach is applied to precisely define the operation requirements of a satellite.

4. Application with PUS standard

This section introduces the PUS Standard and details the "*Telecommand Verification Service*". This service is used as an example in the next section to illustrate the application of CoFI to the OBDH software.

4.1. The PUS Standard

The PUS (*Packet Utilization Standard*) is one of the standards of the ECSS (*European Cooperation for Space Standardization*), released in January 2003. The ECSS is an effort of European national agencies and European industrial associations to develop and maintain common standards. The main benefits of these standards are the costs and efforts reduction regarding conception and development of space missions, ECSS (2003).

The PUS, or the ECSS-E-70-41A standard, focuses on the ground and systems operations related to the utilization of telecommand and telemetry packets. It standardizes these packets and describes sixteen services which the OBDH (*On-Board Data Handling*) should provide. Figure 1 shows these sixteen services. The underlined service is the one used in this work.

Each service has an identification called "*Type*". Depending on the type of the service, there are specific activities, called "*Subtypes*", which are responsible for performing the user´s request. Therefore, the telecommand and telemetry packets are variable: they may correspond to the chosen type and subtype. Figure 2 shows the fields of a telecommand packet, highlighting the field "*Data Field Header*", where the type and subtype are defined in the request. The PUS addresses the shaded fields, although some of the white fields, such as "*Packet ID Type*" and "*Packet ID Data Field Header Flag*", have also been defined in PUS with default values.

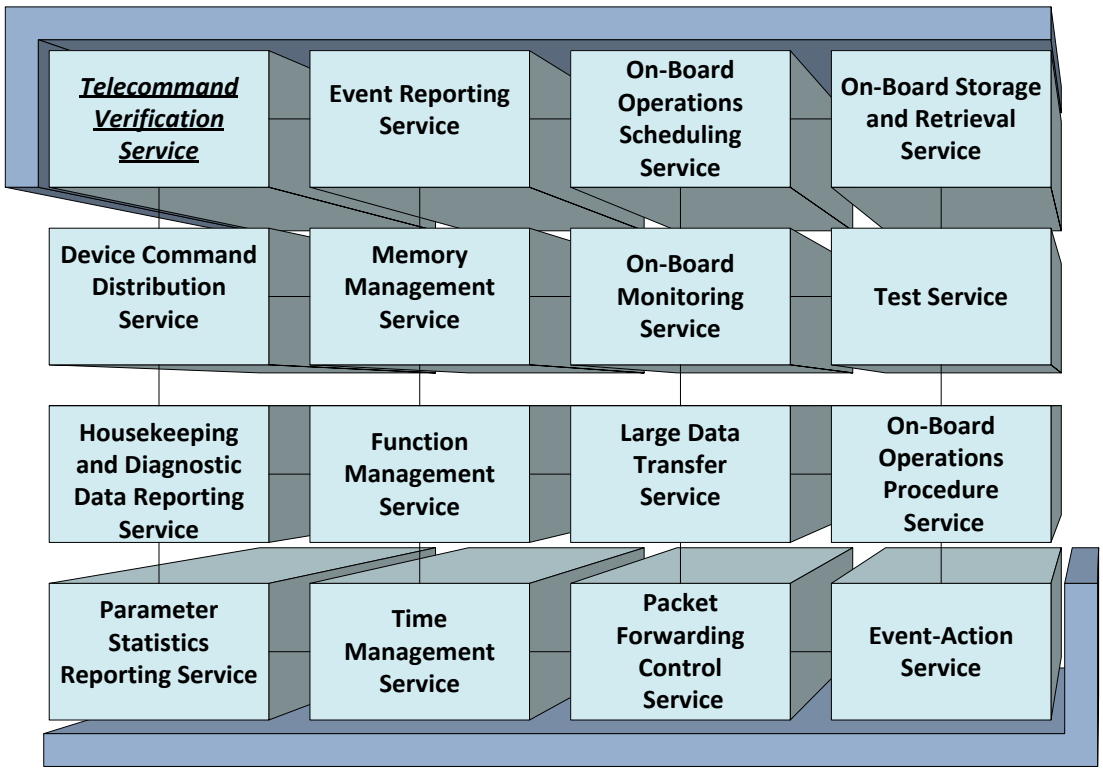


Figure 1. PUS Services.

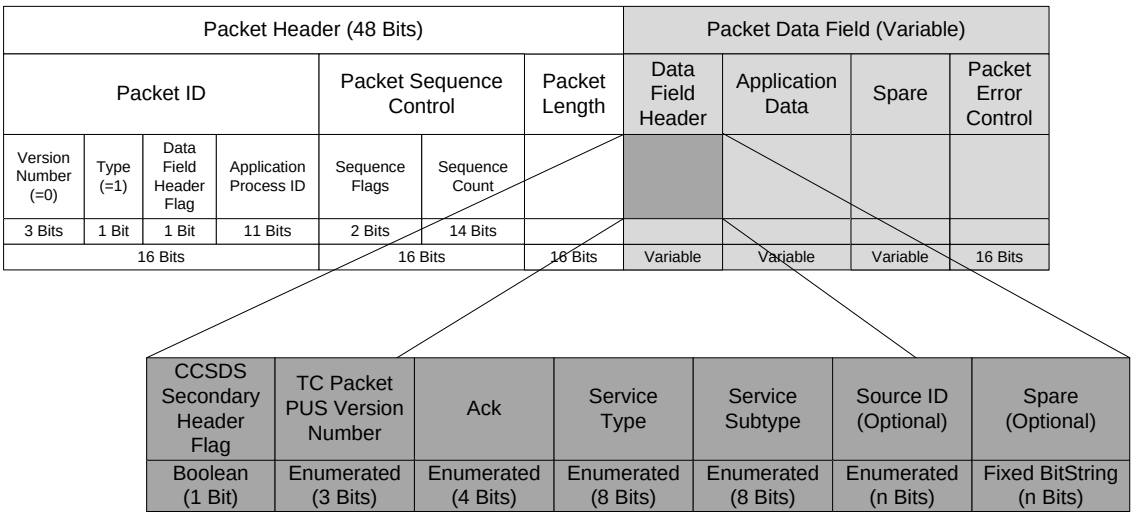


Figure 2. PUS Telecommand Packet.

4.2. Telecommand Verification Service

According to ECSS (2003), The Telecommand Verification Service provides the capability of checking the execution of the each telecommand packet, from its acceptance through to its completion of execution. There are four different stages for the telecommand verification. Although providing the verification of the telecommand, it is not necessary that every telecommand should be verifiable at each stage. The stages are:

- ***Telecommand acceptance***
- ***Telecommand execution started***
- ***Telecommand execution progress***
- ***Telecommand execution completion***

Within the range between the telecommand acceptance and the telecommand completion of execution, the user can request an execution success report, which allows him to follow the exact point of the execution. The success report is requested through the "Ack" field of the telecommand packet. This field is shown in Fig. 2 above.

When a failure occurs at any stage, this service must send a failure report to the user containing the error code and some additional information regarding the cause of this failure. It helps the user to understand the main reason of such failure.

In short, each stage should have two reports: Success Report and Failure Report. It results in eight reports that the telecommand verification service shall provide.

The type number of this service is 1. The subtype numbers of each report are listed in Table 1. For this work, only the "Telecommand Acceptance" and "Telecommand Execution Completion" stages were used.

Table 1. Telecommand verification reports and its subtypes identification.

Report	*(Type , Subtype)*
Telecommand Acceptance Report - Success	***(1 , 1)***
Telecommand Acceptance Report – Failure	***(1 , 2)***
Telecommand Start of Execution Report – Success	***(1 , 3)***
Telecommand Start of Execution Report – Failure	***(1 , 4)***
Telecommand Progress of Execution Report – Success	***(1 , 5)***
Telecommand Progress of Execution Report – Failure	***(1 , 6)***
Telecommand Completion of Execution Report – Success	***(1 , 7)***
Telecommand Completion of Execution Report – Failure	***(1 , 8)***

5. Application of CoFI to Telecommand Verification Service of an OBDH

In this section the telecommand verification service is used as an example to illustrate the application of the CoFI testing methodology to the OBDH software of a satellite that follows the PUS standard. This service is chosen because it is a mandatory service for any OBDH software that follows the PUS. The strategy used in this work is summarized in Fig. 3.

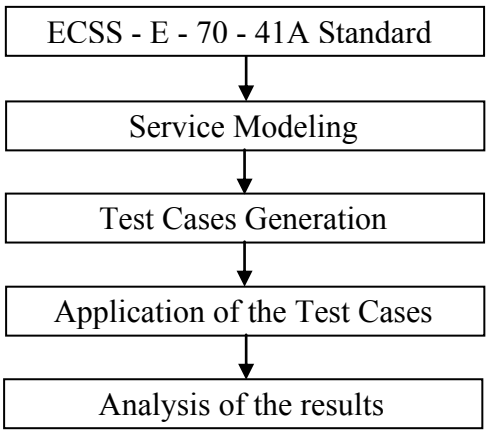


Figure 3. Strategy.

From the description of the service provided by the PUS standard, finite state machines are specified to represent the behavior of specific scenarios. This step uses the CoFI testing methodology to develop the models. It is important to note that this is not done automatically. Then, the test cases are obtained from these state machines using the Condado tool.

The next step is to execute manually the test cases against the OBDH and observe the responses of the OBDH. Both of these activities use a TET (Test Execution

Tool), which is also under development. These steps are shown in Fig. 4. Finally, the errors found with the test cases application are used to analyze the contribution of the testing methodology.

One important point to highlight is that, differently from the works discussed in Section 2, in this work the starting point for the development of the finite state machines is not the requirement specification, but a standard, the PUS, which is used as basis to develop on-board computer software. As a consequence, one of the issues analyzed in this work is the viability of reusing these test cases for any other software that follows this same standard.

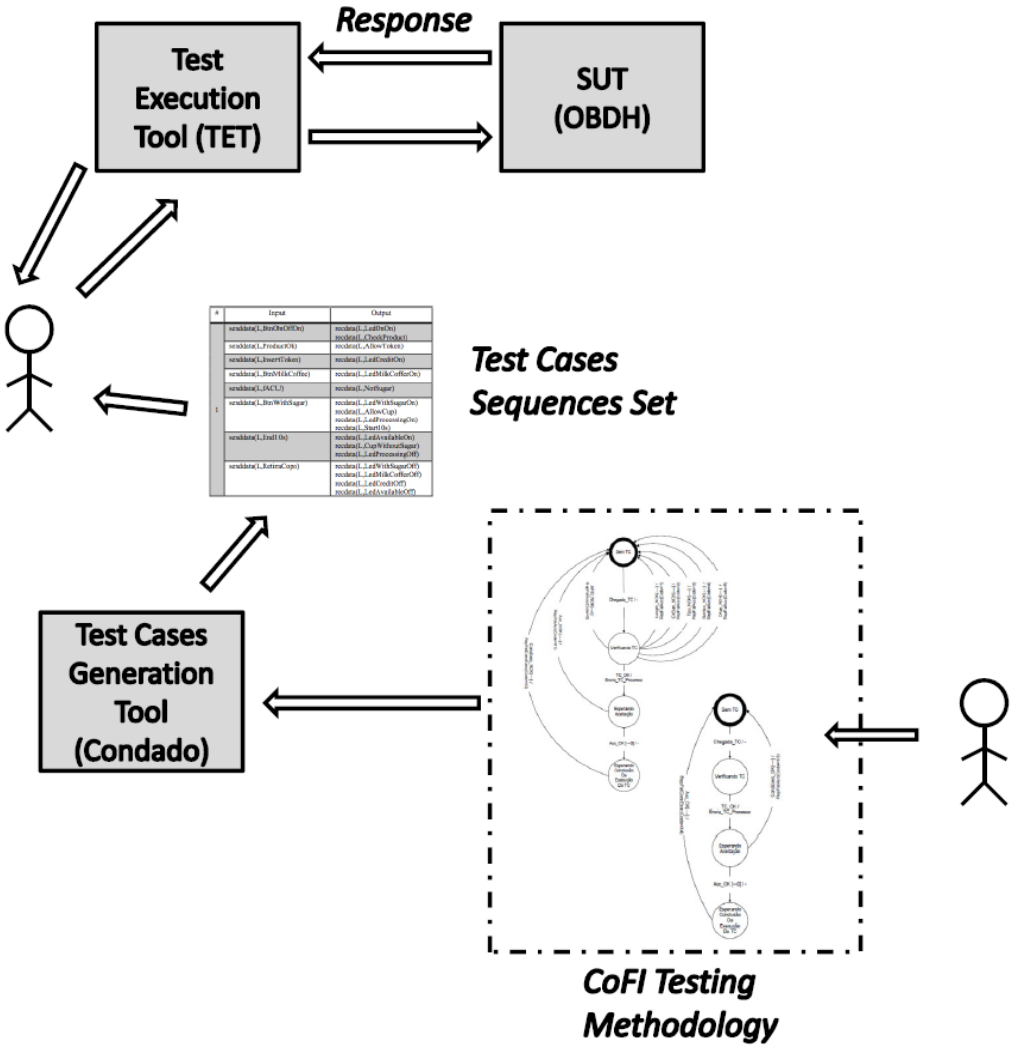


Figure 4. Test application activities.

5.1. Service Modeling in Finite States Machine

After the analysis and understanding of the PUS standard, the telecommand verification service was modeled in finite state machines.

Following the CoFI methodology, four different classes of state machines should be developed: (i) normal, (ii) specified exceptions, (iii) inopportune inputs and (iv)

invalid inputs caused by hardware faults. However, hardware faults are not addressed by the PUS standard. As a consequence, only classes (i) to (iii) could be modeled.

In the case of the telecommand verification service, each class is modeled by one finite state machine. The first model represents the normal behavior of this service and it is shown in Fig. 5. Four states represent the current stage of the service depending on the event associated to the transition. The events "TC_Arrival" and "TC_OK" are events from the embedded software, and thus, the person who is applying the test cases cannot observe their occurrences.

The responses of the system are within the telemetry packets. They contain the reports mentioned in section 4.2. The responses "RepSucAcc" (*Success Report of Telecommand Acception)* and "RepSucCompExec" (*Success Report of Telecommand Completion of Execution)* are only sent if their respective bits, '3' and '0' in the telecommand "*Ack*" field, are set to '1'. Otherwise, they are not sent. The "*Ack*" field of the telecommand is represented by the four bits "0123" inside the brackets in the events "Acc_OK[0123]" and "CompExec[0123]".

Telecommand Verification Service –
Normal Behavior Model

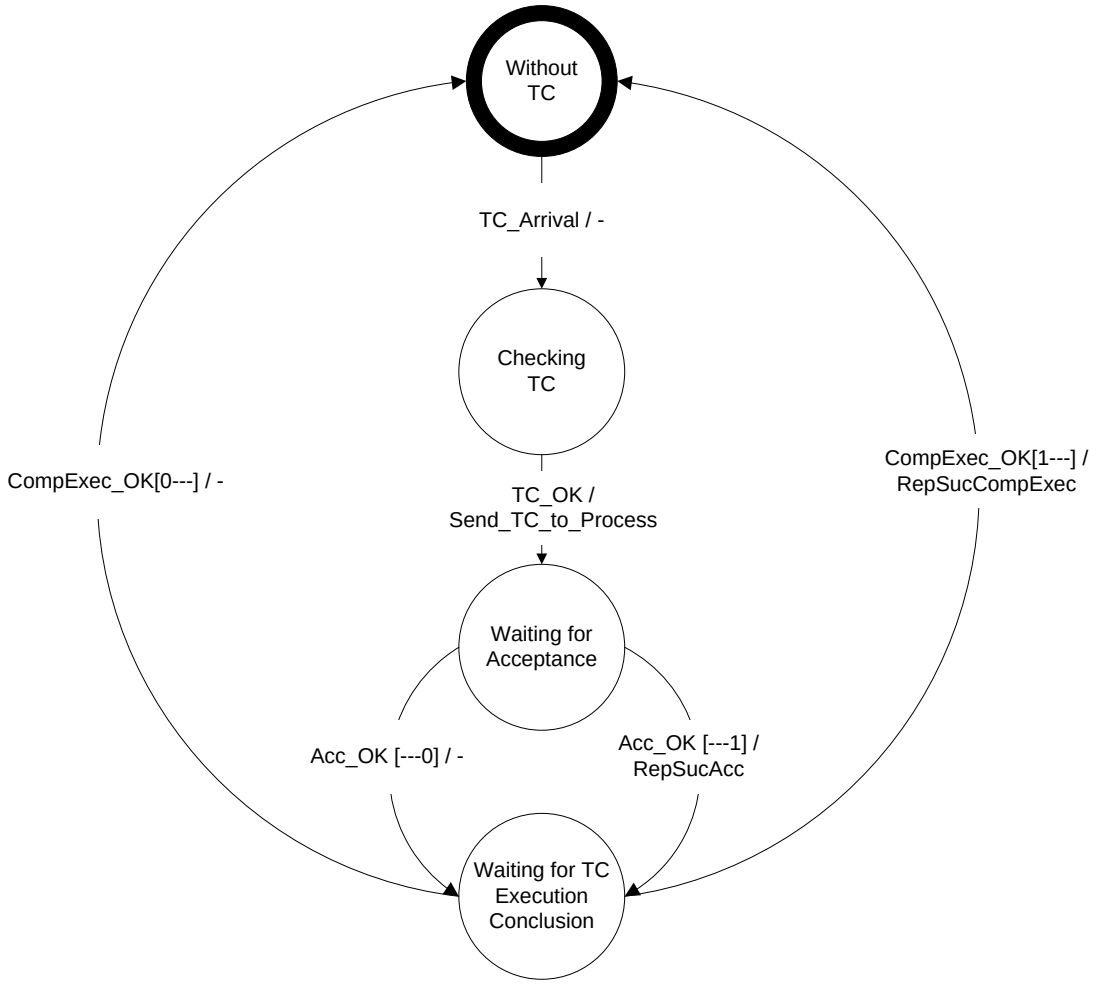


Figure 5. Telecommand Verification Normal Behavior model.

As a second example, the model developed for the Specified Exceptions Behavior is presented in Fig. 6. This class encompasses the events that the standard has defined for specified types of failure or abnormal functioning of the system. In this case, the PUS defines six failures that may occur in the acceptance stage and their respective codes. For other stages, the errors are mission-specific. The codes "X1" and "X2" in Fig.6 show that these errors are not in the standard. Also, it is important to note that the "Ack" bits are represented by this configuration "[----]". It means that the corresponding failure report must be sent regardless the value of these bits.

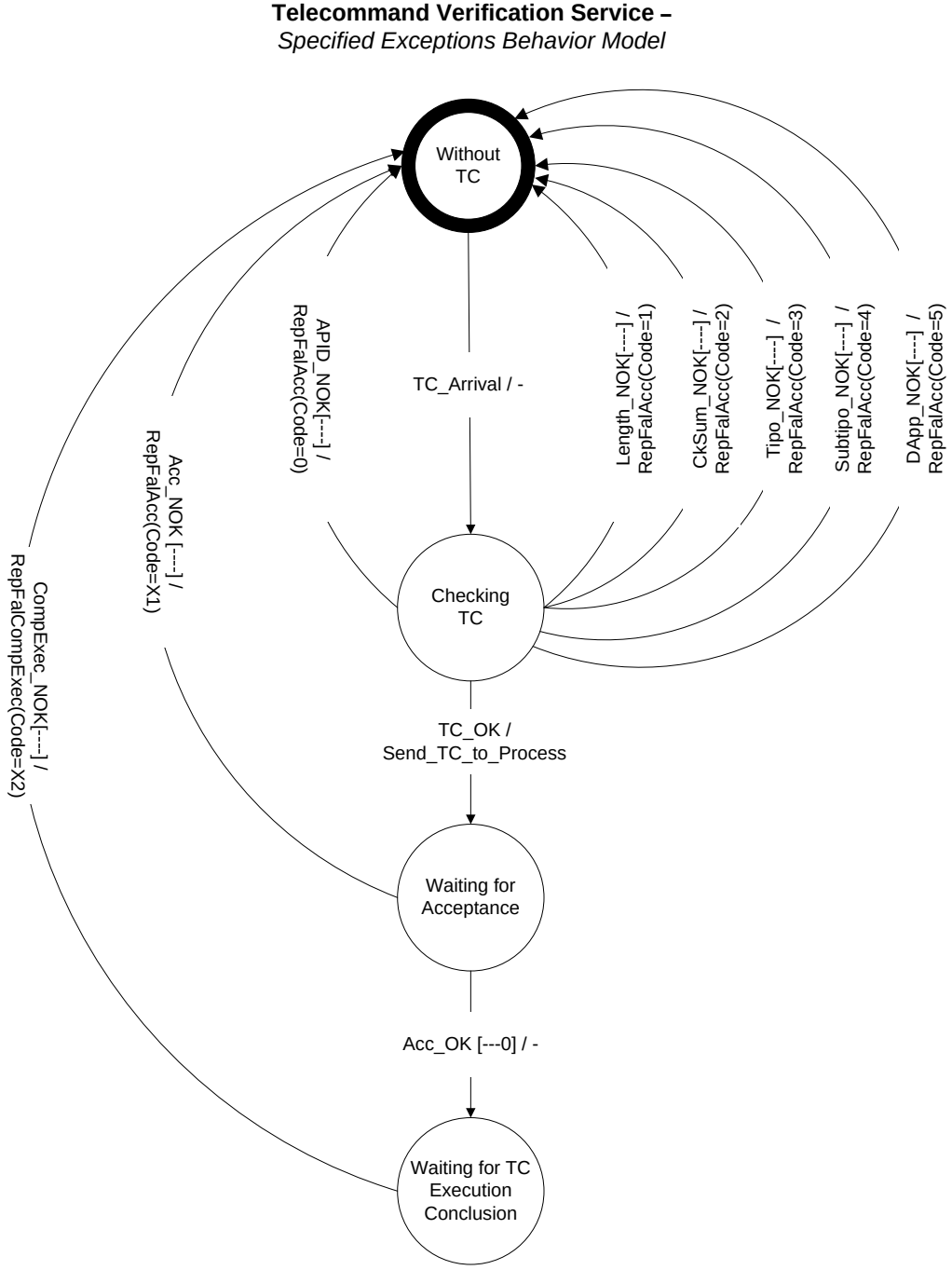


Figure 6. Telecommand Verification Specified Exceptions Behavior model.

The third, and last state machine developed, is the Sneak Paths Behavior, shown in Fig. 7. This model consists in the expected events occurring in inopportune moments. In this model, when the OBDH software is in the acceptance state, waiting for the acceptance event from the process application, if it receives the event "CompExec_OK[----]", the service software shall send a failure report regarding this failure, and return to its initial state. Analogously, when the service software is in execution completion state, if it receives the event "Acc_OK[----]", it shall send the failure report to inform the failure. Note also that, as well as the model of Fig.6, the "*Ack*" field has the configuration "[----]", meaning that regardless its bits values, if the those events occur, the report must be sent and the service software must return to its initial state. The error codes are mission-specific.

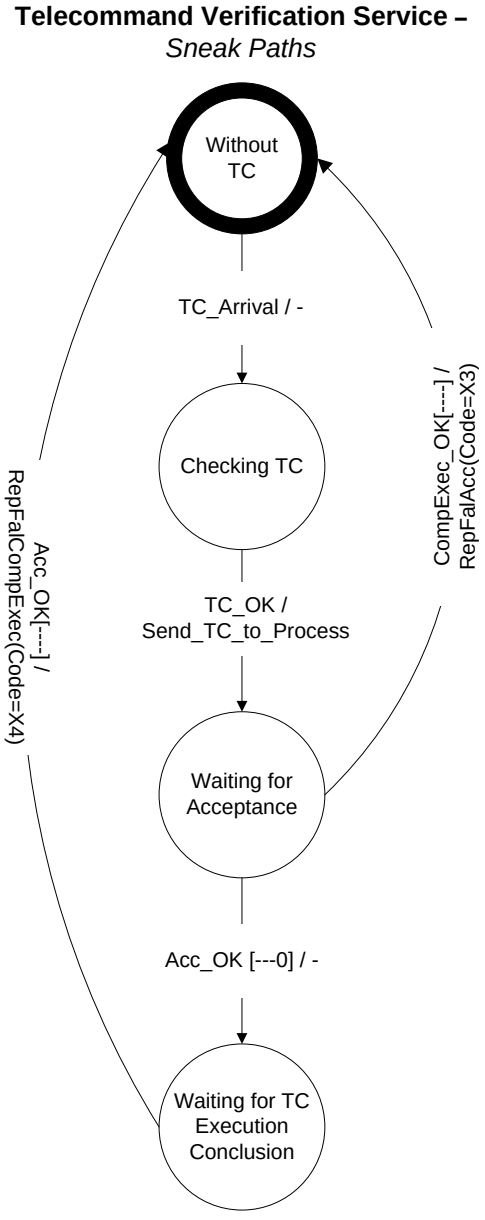


Figure 7. Telecommand Verification Sneak Paths Behavior model.

In general, the size of the finite state machine models is considered small. The biggest model has only four states and eleven transitions.

5.2. Generation of Test Cases

Sixteen test cases were generated from the three developed models. The Tables 3, 4 and 5 show, respectively, some test cases from Normal Behavior Model, Specified Exceptions Behavior Model and Sneak Paths Behavior Model. A test case is a sequence of inputs and outputs.

Table 3. Test cases for Normal Behavior Model.

CASE NUMBER	***INPUT***	***OUTPUT***
2	*TC_Arrival*	-
	TC_OK	*Send_TC_to_Process*
	Acc_OK[---1]	*RepSucAcc*
	CompExec_OK[---0]	-
4	*TC_Arrival*	-
	TC_OK	*Send_TC_to_Process*
	Acc_OK[---1]	*RepSucAcc*
	CompExec_OK[---1]	*RepSucCompExec*

Table 4. Test cases for Specified Exceptions Behavior Model.

CASE NUMBER	***INPUT***	***OUTPUT***
7	*TC_Arrival*	-
	APID_NOK	*RepFalAcc(Code=0)*
8	*TC_Arrival*	-
	Length_NOK	*RepFalAcc(Code=1)*
9	*TC_Arrival*	-
	Checksum_NOK	*RepFalAcc(Code=2)*
10	*TC_Arrival*	-
	Type_NOK	*RepFalAcc(Code=3)*

Table 5. Test cases for Sneak Paths Behavior Model.

CASE NUMBER	***INPUT***	***OUTPUT***
15	*TC_Arrival*	*-*
	TC_OK	*Send_TC_to_Process*
	Acc_OK[---0]	*-*
	CompExec_OK[---1]	*RepFalAcc(Code=X4)*
16	*TC_Arrival*	*-*
	TC_OK	*Send_TC_to_Process*
	CompExec_OK[---1]	*RepFalAcc(Code=X3)*

5.3. Application of Test Cases and Analysis of the Results

The test cases application to the OBDH software was characterized by the following events and results.

Initially, only nine of the test cases were applied to the OBDH software. The other seven test cases could not be applied because the TET did not allow to generate telecommand packets containing the following errors: invalid checksum, invalid packet size and, invalid sequence count. The first consequence of the application of the CoFI testing methodology was a request to modify the TET, improving its flexibility and usability.

As a response to this request, a new version of the TET was generated and three other test cases were applied in a second moment. The remaining four test cases could not be applied because the modeled events are internal to the on-board software. They are related to the communication between the PUS service and the application processes. These events cannot be generated by the TET. These test cases are related to the Sneak Paths Behavior model and basically represent the situations when the application process gives input events to the telecommand verification service in the wrong stages. This functionality may eventually be considered in the future for incorporation in the testability environment of the OBDH.

Regarding the detection of errors in the OBDH software, two test cases resulted in erroneous output. The first one is the application of a test case with one specified exception. In this case, the OBDH software stopped receiving telecommand packets and sent telemetry packets indicating the error code for “acceptance failure”, even if the telecommand packet was a correct one. The second error is related to the reception of two inconsistent telemetry packets. These errors are considered critical, because they can cause the systems’ blockage, compromising the whole mission.

After the presentation of the results, the development team corrected the OBDH software and a new application of the test cases resulted in no error.

The process of modeling, generating and applying the test cases set spent forty hours. This time includes also the time intervals spent by the development team on modifying the TET and the OBDH software, and the second application of the test cases by the testing team.

6. Conclusions

This paper analyzes the contributions of one specific verification technique for the development of space embedded software. The verification technique is the CoFI testing methodology and it is applied to the OBDH software of a satellite that follows the PUS standard.

The main conclusions of this work are the following.

This is a new methodology that is still in its fourth practical experimental application. The results of the utilization of this methodology are being evaluated. Its main limitations are that it does not cover system performance tests, it does not cover tests regarding combination of services, and it does not guarantee the coverage of the code, because it is a black-box testing methodology. However, it has shown itself as a good method to cover tests at system level for acceptance purposes, in which source code is not available.

The CoFI guides the decomposition of the system behavior in different classes of behavior for each different service the system provides. The model of each class of behavior contains only the events related to that class. As a consequence, the models are small. They can be easily understood and analyzed by the development and testing teams. The equivalence of the set of test cases generated from the state machine of the complete system behavior and the set of test cases from the partial models of the system, i.e. smaller state machines, is proven in Ambrosio (2005).

One important contribution of the CoFI methodology is on the specification of the Test Execution Tool (TET). In the case study, the application of CoFI resulted in the elaboration of some requests for providing flexibility of this tool.

Based on the results of the execution of the generated test cases, the relevant contribution of CoFI was in the detection of errors in the OBDH software, which were considered as critical ones. This detection resulted in important corrections of the OBDH software.

All the activities related to the generation of the testing models, the generation of the test cases and the execution of the test cases, described in this paper, was performed by a team independent of the development team. All the models were created based on a standard (the PUS), and not based on the requirement specification of the software under test. This approach shows the reusability of the test cases. The same set of test cases can be applied to any other OBDH software that is based on the same standard.

The next activities are related to the extension of this work to other PUS services, using the same methodology. The On-Board Operations Scheduling Service is being considered due to its complexity.

7. Acknowledgements

The authors would like to thank the collaboration and support of Fabrício Kucinskis, Ronaldo Arias and Thiago Pereira of the Aerospace Electronics Department (DEA) of INPE. The authors would also like to thank the financial support of the Project Sistemas Inerciais para Aplicações Aeroespaciais (SIA), from the FINEP/CTA/INPE.

References

Ambrosio, A. M. (2005) "CoFI – uma abordagem combinando teste de conformidade e injeção de falhas para validação de software em aplicações espaciais", Tese de doutorado, INPE, São José dos Campos(Brazil).

Ambrosio, A. M.; Mattiello-Francisco, M. F; Martins, E. (2008) "An Independent Software Verification and Validation Process for Space Applications", Proceedings of the 9th Conference on Space Operations (SpaceOps). Heidelberg (Germany).

Arias, R.; Kucinskis, F. N.; Alonso, J. D. D. (2008) "Lessons Learned from an Onboard ECSS PUS Object-Oriented Implementation", Proceedings of the 9th Conference on Space Operations (SpaceOps). Heidelberg (Germany).

ECSS – European Cooperation for Space Standardization (2003) "ECSS-E-70-41A – Ground systems and operations: telemetry and telecommand Packet Utilization", Noordwijk: ESA publication Division. Available online in: <http://www.ecss.nl>.

Leveson (2005), "N. Role of Software in Spacecraft Accidents", Journal of Spacecrafts and Rockets, Vol. 41, No. 4, pages 564-575.

Martins, E.; Sabião, S.B.; Ambrosio, A.M. (1999) "ConData: a Tool for Automating Specification-based Test Case Generation for Communication Systems", Software Quality Journal, Vol. 8, No.4, pages 303-319.

Morais, M.H.E; Ambrosio, A.M. (2010) "A new model-based approach for analysis and refinement of requirement specification to space operations", Proceedings of the 10th Conference on Space Operations (SpaceOps). Huntsville (Alabama, USA).

Pontes, R. P. et al. (2009) "A Comparative Analysis of two Verification Techniques for DEDS: Model Checking versus Model-based Testing" ,Proceedings of 4th IFAC Workshop on Discrete Event System Design (DEDes), Spain, pages 70-75.

Detecção e Correção de Falhas Transitórias Através da Descrição de Programas Usando Matrizes

Ronaldo R. Ferreira, Álvaro F. Moreira, Luigi Carro

[1]Instituto de Informática
Universidade Federal do Rio Grande do Sul (UFRGS)
Porto Alegre – RS – Brasil

{rrferreira, afmoreira, carro}@inf.ufrgs.br

Abstract. *Advances in transistor manufacturing have enabled technology scaling along the years, sustaining Moore's law. In this scenario, embedded systems will face restricted resources available to deploy fault-tolerance due the increase of power consumption that these techniques require. In this paper, we claim the detection and correction (D&C) of failures at system level by using matrices to encode whole programs and algorithms. With such encoding, it is possible to employ estabilished D&C techniques of errors occurring in matrices, running with unexpressive overhead of power and energy. We evaluated this proposal using two case studies significant for the embedded system domain. We observed in some cases an overhead of only 5% in performance and 8% in program size.*

Resumo. *Os avanços na fabricação de transistores têm permitido reduzir o tamanho da tecnologia, sustentando a Lei de Moore. Neste cenário, os sistemas embarcados serão projetados com margem reduzida para a implantação de técnicas de tolerância a falhas devido ao aumento no consumo de potência que essas técnicas requerem. Neste artigo, defendemos a detecção e correção (D&C) de falhas em nível de sistema através da codificação de quaisquer programas e algoritmos com matrizes. Essa codificação possibilita empregarmos técnicas estabelecidas de D&C de erros em matrizes, incorrendo em acréscimo inexpressivo de potência e energia. Avaliamos a nossa proposta através de dois estudos de caso relevantes para o domínio de sistemas embarcados, para os quais observamos em alguns casos decréscimo de somente 5% em desempenho e de aumento 8% em tamanho de programa.*

1. Introdução

Os avanços na fabricação de transistores têm permitido reduzir o tamanho da tecnologia, sustentando a Lei de Moore. Com a tecnologia atual de 45 nm amplamente disponível e com as futuras possuindo nodos com tecnologia de 32 nm e 22 nm, as falhas transitórias causadas por radiação gerarão problemas em qualquer produto eletrônico que as utilize. Dado que a distância entre os transistores diminui rapidamente, uma partícula que venha a atingir o núcleo do circuito integrado interferirá em diversas unidades lógicas, acarretando em falhas que perduram durante diversos ciclos de relógio [Lisboa et al. 2007].

Neste cenário, os sistemas embarcados serão requisitados em seus limite de operação; eles deverão oferecer diversos serviços demandando baixíssimo gasto energético e

fornecendo alto desempenho, mesmo na presença de múltiplas falhas. Técnicas clássicas de tolerância a falhas tais como Redundância Modular Tripla (TMR) e Duplicação com Comparação (DwC) impõem acréscimos em área no fator de 3 ou 2 vezes devido a hardware adicional para a implementação dessas técnicas [Huang and Abraham 1984]. Além disso, o alto consumo de potência dessas técnicas as tornam impraticáveis para serem implantadas no domínio de sistemas embarcados [Argyrides et al. 2009].

Em um sistema no qual a disponibilidade de potência e energia são escassas, a solução mais viável para tratar falhas transitórias é detectá-las e corrigi-las em software, em nível de sistema [Argyrides et al. 2009, Huang and Abraham 1984, Pattabiraman et al. 2007, Vemu et al. 2007, Oh et al. 2002]. Deixar essa tarefa para o desenvolvedor é impraticável, pois acarretaria em acréscimos elevados nos tempos de desenvolvimento e teste do software, além na complexidade do mesmo, o que comprometeria o *time-to-market*. Portanto, o mecanismo de tolerância a falhas deve estar incorporado na linguagem de programação, implementado pelo seu compilador ou interpretador. Assim, exime-se o desenvolvedor de software de tratar essas falhas, beneficiando o processo de desenvolvimento e reduzindo o *time-to-market*.

Neste trabalho apresentamos os fundamentos essenciais para se realizar detecção e correção de falhas transitórias utilizando construções algébricas formais, adotando matrizes como maneira de se descrever quaisquer algoritmos e programas, e propomos também a incorporação desse formalismo em uma linguagem de programação. Linguagens baseadas em matrizes, por exemplo, Matlab, são amplamente adotadas pela indústria de sistemas embarcados. Apresentamos aqui esses fundamentos através de dois algoritmos importantes para o domínio de software embarcado: geração do código de Huffman [Huffman 1952] e as transformadas MDCT (Modified Discrete Cosine Transform) e IMDCT (Inverse MDCT) [Princen et al. 1987]. Por fim, apresentamos como podemos eficientemente detectar e corrigir falhas transitórias usando somente operações sobre matrizes, fornecendo suporte para a implementação da técnica proposta.

A contribuição principal deste trabalho é um mecanismo para detectar e corrigir falhas em nível de sistema, o qual incorre em acréscimos mínimos em área, desempenho e potência. Alcançamos esse objetivo através da definição de programas e algoritmos completamente com matrizes; sobre essas é possível utilizar técnicas de verificação de baixo gasto energético. Os resultados experimentais dos dois estudos de caso radicalmente diferentes ilustram os princípios do método e sua generalidade para outras aplicações.

Este texto está organizado da seguinte maneira: a seção 2 discute os trabalhos relacionados; a seção 3 revisa a técnica utilizada para detectar e corrigir erros em matrizes; a seção 4 apresenta os estudos de caso e introduz a matemática necessária para se expressar programação dinâmica com operações sobre matrizes; a seção 5 apresenta os resultados experimentais usados para demonstrar a viabilidade da técnica proposta; e a seção 6 discute as conclusões e os trabalho futuros.

2. Trabalhos Relacionados

Os autores em [Blum et al. 1989] provaram que para qualquer anel parcialmente ordenado, existe uma máquina universal correspondente sobre esse anel. Como apresentaremos na seção 4.1, os autores em [Atallah et al. 1989] descrevem qualquer algoritmo baseado em árvores através de um semi-anel de matrizes. Portanto, caso exista um anel

parcialmente ordenado de matrizes, sua máquina correspondente, de acordo com o modelo de Blum [Blum et al. 1989], é universal, bem como a nossa abordagem. A construção de um anel para qualquer tipo de algoritmo, não somente os baseados em árvores e os naturalmente representáveis por matrizes, é um dos nossos trabalho futuros. Este artigo estuda a viabilidade prática de uma abordagem completamente baseada em matrizes.

Tolerância a falhas baseada em algoritmo (ABFT) [Huang and Abraham 1984] refere-se ao caso no qual a técnica de tolerância a falhas é incorporada à computação dos dados. Os autores em [Huang and Abraham 1984] utilizaram ABFT para corrigirem operações sobre matrizes baseados em *checksums*. Os autores enfatizaram que para se utilizar ABFT a operação sendo protegida deve obrigatoriamente preservar a propriedade do *checksum*, o que não é sempre o caso. Eles detectam e corrigem vários elementos com erro na matriz resultante, mas dependem para tal de uma rede inter-conectada com diversos processadores. Neste trabalho, a detecção e a correção são independentes das operações e da organização do hardware.

Verificação de programas [Blum and Kanna 1989] é uma técnica utilizada para verificar se os resultados produzidos por um programa estão corretos ou não. Para que essa técnica seja viável, o mecanismo de verificação deve ser assintoticamente menor que o algoritmo sendo verificado. Do contrário, a verificação de programas equivale-se à recomputação dos resultados. os autores em [Prata and Silva 1999] mostraram, com o suporte de campanhas de injeção de falhas, que verificação de programas é superior à ABFT, incorrendo em acréscimos consideravelmente menores no tempo total de execução do programa.

Os autores em [Lisboa et al. 2007] propuseram utilizar o vetor da técnica de Freivalds [Motwani and Raghavan 1995] fixado em $r = \{1\}^n$, protegendo com essa técnica um multiplicador de matrizes implementado em hardware. Em [Argyrides et al. 2009], o esquema apresentado em [Lisboa et al. 2007] foi estendido para permitir a correção de erro em um elemento da matriz resultante. Este trabalho adota essas técnicas como mecanismo de correção e detecção de erros em matrizes, implementando-as em software.

A proteção de variáveis críticas de uma aplicação pode ser realizada através de análise estática durante a compilação do código-fonte [Pattabiraman et al. 2007]. Nesse método, realiza-se o particionamento do programa a partir dos blocos básicos dele extraídos, seguindo a análise de criticidade de uma variável realizada através da contagem de leituras e escritas de cada variável. Os autores em [Pattabiraman et al. 2007] obtiveram um acréscimo médio de 33% em tempo total de execução para proteger somente 5 variáveis. Comparada à nossa abordagem, essa taxa é altíssima. Considerando uma variável de 32 bits, através da análise de criticidade de cada variável, há um acréscimo médio de 33% para se proteger somente 160 bits, enquanto na nossa abordagem protegemos uma matriz composta de n^2 e $n \times n/2$ variáveis com menos penalidade ao desempenho, sendo de 5% para Huffman e de $\sim$ 30% para MDCT/IMDCT.

Outro problema causado nas aplicações devido a falhas transitórias é a inserção de falhas no fluxo de controle. Os autores em [Vemu et al. 2007] apresentam um mecanismo - batizado ACCE - baseado em software para a detecção e correção desse tipo de falha. Após a análise de dependência entre os blocos básicos, ACCE é capaz de detectar o nodo de controle que contém erro no grafo de execução, permitindo sua posterior

correção. Neste trabalho, assume-se que as falhas de controle são tratados em hardware, através da triplicação do contador de programa. Nesse caso, adicionam-se somente dois registrados, incorrendo em acréscimo em área negligenciável. Além disso, este trabalho propõe a descrição completa da aplicação com matrizes, o que acaba com as asserções de controle na aplicação, amortizando a ocorrência de falhas de controle. Esse mecanismo de correção de falhas de controle em matrizes é um dos nossos trabalhos futuros.

Outra técnica para detectar falhas transitórias é o uso de invariantes de software através da detecção automática de pré-, pós-condições e invariantes de laço. Esse método baseia-se em ferramentas estado-da-arte para a detecção de invariantes, tais como a Daikon [Ernst et al. 2007]. Infelizmente, sendo a detecção de invariantes de laço um problema indecidível, essas ferramentas empregam heurísticas para inferir alguns dos invariantes. Alguns autores reportam resultados bastante negativos ao se usar invariantes para detectar falhas [Krishnamurthi and Reiss 2003]. Pode-se melhorar a detecção de falhas com invariantes, mas, geralmente, necessitando de procedimentos de instrumentação de código [Cheynet et al. 2000]. Apesar do baixo acréscimo no tempo de execução incorrido pela técnica de invariantes, [Lisboa et al. 2009], a taxa de detecção de falhas é baixa comparada à técnica de matrizes [Lisboa et al. 2007]. Isso ocorre porque os detectores de invariantes não são cientes da semântica implementada na aplicação, pois elas trabalham sobre o código-fonte.

Os autores em [Chen and Kandemir 2005] propuseram uma Máquina Virtual Java (JVM) tolerante a falhas para o domínio de sistemas embarcados. Nessa JVM, há dois motores executando duas instâncias da aplicação, trabalhando de maneira similar à duplicação com comparação. Para amortizar o alto acréscimo no tamanho de programa imposto pela duplicação de objetos, os autores propuseram um mecanismo de compartilhamento de objetos entre as duas instâncias da aplicação em execução. Essa abordagem sempre incorre em um acréscimo maior que 100% no tempo total de execução, ao executar em ambientes monoprocessados. Na nossa abordagem, o acréscimo pode ser tão baixo quanto 5% em alguns casos. Existem linguagens de programação tolerantes a falhas com o objetivo de recuperar automaticamente a execução em sistemas distribuídos na camada de aplicação [Florio and Blondia 2008], mas nenhuma dessas é voltada ao domínio de sistemas embarcados, no qual o desenvolvimento de software possui restrições severas em recursos de hardware.

3. Fingerprinting de Matrizes

Fingerprinting consiste em verificar se x e y são iguais dado um universo U. Considerando um mapeamento aleatório de U em um universo V, onde $|V| \ll |U|$, pode-se verificar eficientemente que $x = y$ se e somente se suas imagens são as mesmas em V. Esse mapeamento em um outro universo de menor cardinalidade reduz o espaço de busca, tornando o processo de verificação mais eficiente do que recomputar a operação em U [Motwani and Raghavan 1995].

A *técnica de Freivalds* tal como apresentada em [Motwani and Raghavan 1995] verifica se a multiplicação de matrizes $XY = Z$ em tempo $O(n^2)$, sendo muito mais eficiente do que o melhor algoritmo conhecido para a multiplicação de matrizes, o qual é $O(n^{2,376})$ [Coppersmith and Winograd 1987]. Apesar da existência desse algoritmo, a maioria das implementações adotam a solução trivial de tempo $O(n^3)$ devido a sua

clareza e concisão de codificação. Na realidade, a técnica de Freivalds é capaz de verificar qualquer identidade entre matrizes $X \bullet Y = Z$, mas é equivalente à recomputar $X \bullet Y$ quando a operação $\bullet$ é $O(n^2)$, o que é o caso da adição e subtração de matrizes. A técnica de Freivalds se caracteriza pelo seguinte teorema provado em [Motwani and Raghavan 1995]:

Teorema 1 *Seja $XY \neq Z$ e sejam X e Y matrizes $n \times n$. Escolha aleatoriamente de maneira uniforme um vetor $r \in \{0,1\}^n$. Tem-se $X(Yr) = Zr$ com probabilidade $p \leq 1/2$.*

A computação de $X(Yr)$ e Zr requer $O(n^2)$ para cada operação. Portanto, a verificação da identidade de matrizes reduz-se à verificação se os dois vetores calculados são iguais, e essa operação pode ser realizada em tempo $O(n)$. Assim, a operação total requer $O(n^2)$. Os autores em [Lisboa et al. 2007] adotaram a técnica de Freivalds fixando $r = \{1\}^n$. De acordo com o Teorema 1, esse vetor é uma das possibilidades que pode ser escolhida aleatoriamente. Ao fixar r como proposto, tem-se o seguinte corolário provado em [Lisboa et al. 2007]:

Corolário 1 *Seja $XY \neq Z$ e sejam X e Y matrizes $n \times n$. Fixe o vetor $r = \{1\}^n$. Nesse caso, tem-se $X(Yr) \neq Zr$ com probabilidade 1.*

Fixando r como apresentado no Corolário 1 nos permite utilizar a técnica de Freivalds como um verificador eficiente de operações sobre matrizes. Em [Argyrides et al. 2009], estendeu-se Freivalds para permitir a correção e detecção do elemento errôneo da matriz resultante Z com somente duas adições, o que requer tempo $O(1)$. Neste trabalho, adotamos a técnica de Freivalds como apresentado no Corolário 1, aplicando-a para todas as operações sobre matrizes ocorrendo nos programas, incluindo as que requerem tempo $O(n^2)$ para executar. Também adotamos o mecanismo de correção proposto em [Argyrides et al. 2009], oferecendo-nos um *framework* de correção extremamente eficiente no caso de se detectar um erro após a computação de Freivalds e da checagem da identidade $X \bullet Y = Z$. No caso de erro, evita-se a recomputação de $X \bullet Y$, reduzindo a complexidade de $O(n^3)$ e $O(n^2)$ para $O(1)$, caso a operação $\bullet$ seja $O(n^3)$ e $O(n^2)$ respectivamente.

4. Estudos de Caso

Apresentamos nessa seção os dois algoritmos utilizados como estudo de caso nos experimentos de injeção e proteção contra falhas. Huffman e MDCT/IMDCT são algoritmos importantes, sendo componentes centrais de aplicações multimídia, portanto, são claramente exemplos representativos de aplicações embarcadas reais.

4.1. Código de Huffman

Essa seção apresenta o algoritmo que calcula a árvore ótima de Huffman baseado inteiramente em multiplicações e adições de matrizes definidas sobre um semi-anel. Esse e outros algoritmos foram apresentados em [Atallah et al. 1989], onde problemas de programação dinâmica foram reduzidos à operações de matrizes definidas em semi-anéis e, como apresentado, essa abordagem é aplicável a uma ampla classe de problemas de programação dinâmica.

Sejam $(p_1, \ldots, p_n)$ o vetor ordenado de freqüências do alfabeto, $p_{i,j} = \sum_{k=i}^{j} p_k$ a freqüência acumulada entre as palavras i e j do alfabeto, e S uma matriz $n \times n$ contendo

freqüências acumuladas como segue:

$$S_{i,j} = \begin{cases} p_{i+1,j} & \text{se } i < j \\ +\infty & \text{se } i \geq j \end{cases} \quad (1)$$

Esse algoritmo é definido sobre o semi-anel {min, +} (conhecido como semi-anel *tropical*). Esse semi-anel possui $\mathbb{R} \cup \{+\infty\}$ como seu domínio e $\forall a, b \in \mathbb{R} \cup \{+\infty\}, a \oplus b = min(a, b)$ e $a \otimes b = a + b$. No semi-anel tropical, $\forall a \in \mathbb{R} \cup \{+\infty\}, a + \infty = (+\infty) + a = +\infty$ e $min(a, +\infty) = a$. Seja A_h a matriz contendo todos os comprimentos de árvores de Huffman, na qual a entrada $(A_h)_{i,j}$ contém o comprimento de caminho das árvores $(p_{i+1}, \ldots, p_n)$ de tamanho no máximo h, com $0 \leq h \leq \lceil \log n \rceil$. Com essas definições e sendo $0 < i < j \leq n$, A_h é uma matriz $n \times n$ definida recursivamente da seguinte maneira:

$$(A_0)_{i,j} = \begin{cases} +\infty & \text{se } i \geq j \text{ ou } (j - i) > 1 \\ 0 & \text{caso contrário} \end{cases} \quad (2)$$

$$A_h = A_{h-1} \oplus \Big(A_{h-1} \otimes (A_{h-1} + S) \Big) \quad (3)$$

Onde as operações tropicais $X \otimes Y$ e $X \oplus Y$ são:

$$(X \otimes Y)_{i,j} = \bigoplus_{k=1}^{n} X_{i,k} \otimes Y_{k,j} \quad (4)$$

$$(X \oplus Y)_{i,j} = X_{i,j} \oplus Y_{i,j} \quad (5)$$

Na Equação 2, o 0 vem do fato que o comprimento de caminho da árvore até a altura $h = 0$ é igual a 0. Note que na Equação 3 a operação + é a adição tradicional de matrizes, não a definida pelo semi-anel tropical. Note também que a Equação 4 é similar à multiplicação tradicional de matrizes, requerendo tempo $O(n^3)$, e a Equação 5 é similar à adição tradicional de matrizes, requerendo tempo $O(n^2)$. A entrada $(A_{\lceil \log n \rceil})_{1,n}$ contém o comprimento de caminho ótimo da árvore de Huffman de n folhas. Isso conclui a apresentação da geração do código de Huffman implementado totalmente com adição e multiplicação de matrizes. Ressalta-se que essa redução de programação dinâmica em operações sobre o semi-anel tropical aplica-se a qualquer algoritmo baseado em árvores, não somente para Huffman [Atallah et al. 1989].

4.2. Transformadas MDCT e IMDCT

Duas funções importantes no domínio de aplicações multimídia são as transformadas MDCT e IMDC. Essas funções são amplamente utilizadas em aplicações multimídia de tempo-real em padrões de áudio e vídeo, tais como MPEG e MP3. Tanto a MDCT quanto a IMDCT são calculadas de maneira bastante simples, sendo implementadas inteiramente com multiplicação de matrizes representando os dados em compressão. Os algoritmos usados neste artigo foram extraídos de [Cheng and Hsu 2003].

Sejam x^T e $\hat{x}^T$ dois vetores transpostos de comprimento n, os quais representam os valores originais e os produzidos pela IMDCT, respectivamente. Sejam M uma matriz

$n/2 \times n$ contendo os coeficientes da transformada e X um vetor de comprimento $n/2$ contendo o resultado da MDCT. A MDCT e a IMDCT podem ser calculadas como segue:

$$X = Mx \tag{6}$$

$$\hat{x} = M^T X \tag{7}$$

Onde $M_{i,j}$, com $0 \leq i \leq n/2$ e $0 \leq j \leq n-1$, é:

$$M_{i,j} = \cos\left[\frac{\pi}{2n}\left(2j + 1 + \frac{n}{2}\right)(2i+1)\right] \tag{8}$$

As Equações 6 e 7 são computadas e verificadas em tempo $O(n^2)$. Nesse caso, a MDCT e a IMDCT se beneficiarão de alguma forma da técnica de Freivalds ao se proteger o resultado da multiplicação e ao recomputar a matriz em caso de erro. Entretanto, o acréscimo no tempo de execução ao proteger as transformadas não será tão eficientemente amortizado ao aumentar o tamanho do espaço do problema como em Huffman. A próxima seção discute esses aspectos de amortização da proteção e espaço do problema.

5. Experimentos e Resultados

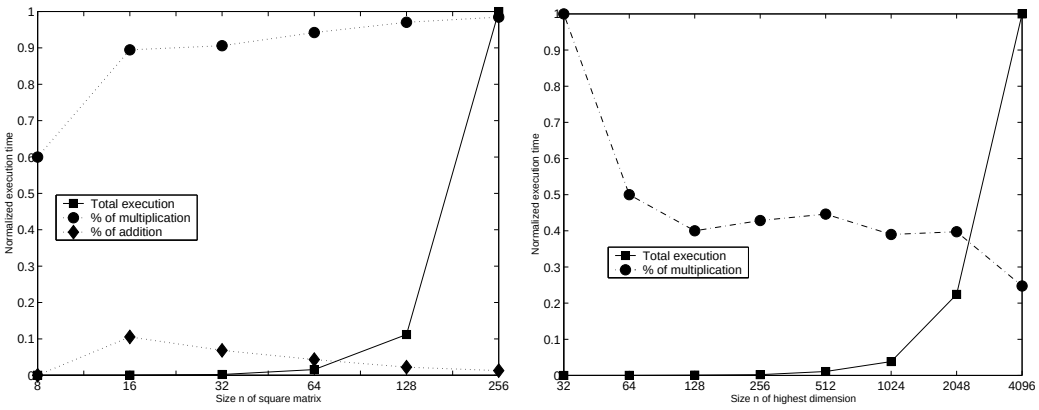
5.1. Metodologia

Utilizamos um desktop Intel Dual Core 1.6 GHz com 1 GB de memória RAM, executando o sistema operacional Windows XP com Service Pack 3 e Matlab R11.1. O Matlab foi configurado para executar em somente um processador, permitindo extrair resultados mais apurados. Mensuramos o tempo de execução para cada estudo de caso através do mecanismo interno do Matlab de *profiling*. A carga do processador foi mantida a mais constante possível entre as mensuráveis. As entradas foram geradas com a função interna *rand* do Matlab, evitando enviesar a execução e alterar o desempenho real devido a entradas pré-determinadas.

Codificamos as seguintes operações sobre matrizes: multiplicações e adições tradicional e tropical. Não utilizamos as funções internas de adição e multiplicação de matrizes do Matlab para evitar viés de execução e otimizações, permitindo uma comparação justa com as operações definidas sobre o semi-anel tropical. Nos experimentos consideramos o tamanho mínimo da matriz aquele para o qual o tempo de execução calculado pelo Matlab tenha sido maior que 0, com precisão de 0,016 segundos. Realizamos injeção de falhas através da troca de valores de um dos elementos de cada matriz computada durante as iterações dos algoritmos, sendo necessária a detecção e a correção desse elemento. Por exemplo, em Huffman para cada altura de árvore h sendo computada, são calculadas três matrizes, sendo inseridos, portanto, três falhas a cada iteração.

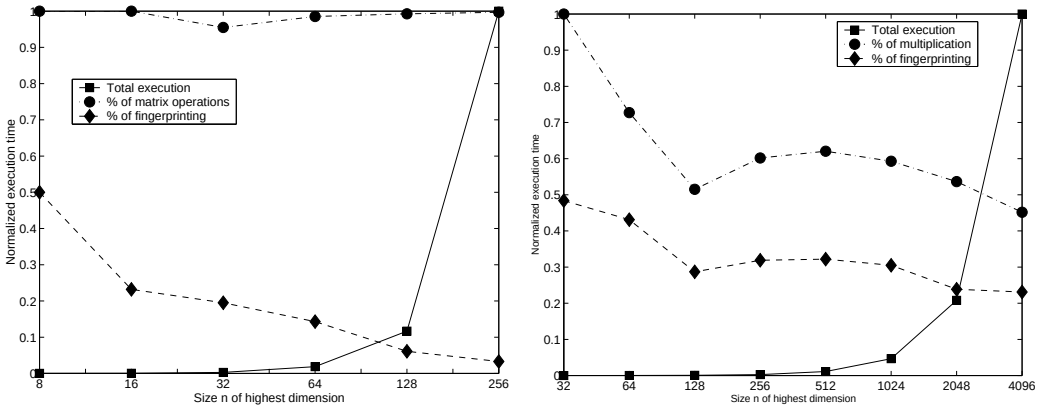
5.2. Resultados

Mensuramos a influência das operações sobre matrizes no tempo total de execução das aplicações, *i.e.* a porcentagem do tempo de execução exclusivamente gasto com a computação de adições e multiplicações de matrizes, tanto as tradicionais quanto as tropicais. Fez-se também a comparação do ganho em desempenho da nossa abordagem em relação à duplicação com comparação. As Figuras 1, 2 e 3 apresentam os resultados.



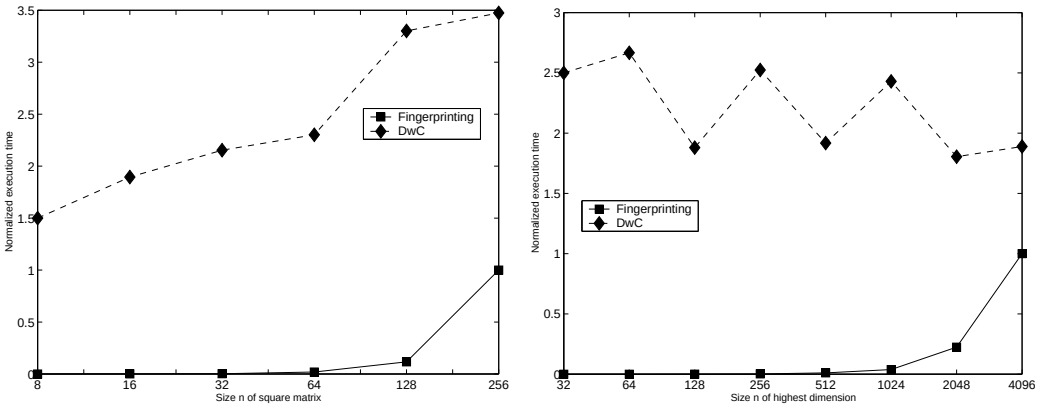
(a) Huffman (b) MDCT/IMDCT

Figura 1. Influência de operações sobre matrizes no tempo total de execução para (a) Huffman e (b) MDCT/IMDCT



(a) Huffman (b) MDCT/IMDCT

Figura 2. Acréscimo no tempo de execução devido a fingerprinting para (a) Huffman e (b) MDCT/IMDCT.



(a) Huffman (b) MDCT/IMDCT

Figura 3. Acréscimo no tempo de execução de duplicação com comparação (DwC) relativo a fingerprinting para (a) Huffman e (b) MDCT/IMDCT.

5.2.1. Influência das Operações em Matrizes sobre o Tempo Total de Execução

A Figura 1 apresenta a influência das operações de matrizes sobre o tempo total de execução das aplicações. A Figura 1(a) apresenta que, como esperado para o algoritmo de Huffman introduzido na seção 4.1, $\sim 100\%$ do tempo total de execução é dedicado à operações de adição e multiplicação de matrizes. Quando $n \geq 16$, a mulplicação de matrizes compreende mais de 90% do tempo total de execução; esses resultados mostram que Huffman se beneficiará muito da técnica de Freivalds. Quando $n = 8$, a porcentagem de multiplicação de matrizes é $\sim 60\%$. Nesse caso, a multiplicação de matrizes não é amortizada eficientemente em relação ao tempo necessário para calcular as matrizes S e A, como apresentado nas Equações 1, 2, e 3. Portanto, para o caso do algoritmo de Huffman (e qualquer outro descrito com o semi-anel tropical), a influência das operações sobre matrizes aumenta significativamente com o tamanho do espaço de problema.

Por outro lado, a Figura 1(b) mostra que para as aplicações MDCT/IMDCT, a influência de operações sobre matrizes no tempo total de execução diminui com o aumento do tamanho do espaço de problema. Quando esse aumenta, o tempo dedicado à computação do cosseno na Equação 8 é consideravelmente maior que a computação das Equações 6 e 7. Entretanto, a multiplicação de matrizes compreende $\sim 40\%$ do tempo total de execução quando $128 \leq n \leq 2048$, estando longe de ser negligenciável. Note que nas transformadas MDCT/IMDCT a única operação de matrizes existente é a multiplicação.

5.2.2. Diminuição do Desempenho devido ao Fingerprinting de Operações em Matrizes

Mensuramos o acréscimo incorrido ao proteger as operações sobre matrizes com fingerprinting, relacionando o tempo de execução gasto em fingerprinting com o tempo gasto em operações sobre matrizes. A Figura 2 apresenta os resultados. No caso de Huffman, implementamos a técnica de Freivalds usando a multiplicação e a adição tropical sem nenhuma alteração em Freivalds. Nesses experimentos, introduzimos um valor errôneo em cada matriz computada durante a execução de Huffman. Lembre que fingerprinting consiste em detectar a existência do valor errôneo e prosseguir com a sua correção. Essas operações são realizadas em tempo $O(n^2)$ e $O(1)$, respectivamente.

A Figura 2(a) apresenta a mensuração para Huffman. A linha pontilhada superior com círculos é a porcentagem do tempo total de execução (o qual é representado pela linha sólida com quadrados) gasto com operações sobre matrizes para uma dada dimensão n. A linha tracejada com diamantes representa a porcentagem do tempo total de execução gasto com fingerprinting. Esses resultados mostram que os custos de se realizar fingerprinting são amortizados com o aumento do tamanho da matriz, sendo de 5% quando $n = 256$ e $\sim 20\%$ quando $16 \leq n \leq 64$.

Esses resultados são muito promissores, dado que a indústria atualmente enfrenta o rápido acréscimo no volume de dados tratados em sistemas embarcados comuns e em eletrônica de consumo. Nesse sentido, pode-se esperar que as matrizes manipuladas possuirão tamanhos grandes, o que amortiza o custo de realizar fingerprinting na aplicação. Note que é possível realizar fingerprinting de maneira mais eficiente: o algoritmo de multiplicação de matrizes utilizado neste trabalho possui complexidade $O(n^3)$, mas, como

apresentando anteriormente, há uma alternativa de complexidade $O(n^{2,376})$.

Obtemos resultados similares para as transformadas MDCT/IMDCT, como apresentado na Figura 2(b). O percentual de acréscimo no tempo total de execução obtido foi de $\sim 30\%$ para $128 \leq n \leq 2048$. Como discutido anteriormente, o cálculo do cosseno demanda $\sim 60\%$ do tempo total de execução, o que faz que a técnica de fingerprinting tenha gasto de tempo percentual linear com o aumento de n. Isso é explicado pelo fato de as Equações 6 e 7 operarem sobre vetores. Assim, o custo de execução aumenta quadraticamente em relação a n, não cubicamente como em Huffman.

5.2.3. Desempenho de Fingerprinting e de Duplicação com Comparação

Mensuramos o acréscimo em tempo de execução de Duplicação com Comparação (DwC) relativamente a fingerprinting. Em ambas as técnicas, injetamos um elemento errôneo em cada matriz computada pelos algoritmos, sendo necessário corrigir esse elemento. Para DwC, calculou-se duas vezes a função, comparou-se seus valores de retorno e, se diferentes, computou-se essa função pela terceira vez. A Figura 3 apresenta os resultados.

Devido à alta influência das operações de matrizes no tempo total de execução de Huffman, DwC é muito inferior em relação a desempenho quando comparado a Freivalds, como pode ser observado na Figura 3(a). Quando $16 \leq n \leq 64$, o acréscimo no tempo de execução incorrido por DwC é $\sim 200\%$, podendo alcançar até 300% para $n \geq 128$ quando comparado proporcionalmente a fingerprinting.

Apesar do alto acréscimo no tempo total de execução ao se proteger as transformadas MDCT/IMDCT com DwC, esse permaneceu constante com o acréscimo em tamanho do problema, como pode ser visto na Figura 3(b). O acréscimo oscilou entre 200% e 250% do tempo total de execução relativamente a fingerprinting, o que nos mostra a superioridade da técnica de fingerprinting em termos de desempenho, dando suporte à abordagem proposta neste trabalho.

5.2.4. Análise do Tamanho Total de Programa

Mensuramos o acréscimo em tamanho de programa imposto pelo uso de matrizes e de Freivalds. Para tal, codificamos Huffman em C e o compilamos com o *gcc* com a opção de otimização de código *-O3*, tendo como alvo a arquitetura x86. Para mensurar o tamanho de cada função codificada, criamos um arquivo .c para cada função compondo cada algoritmo, e utilizamos a ferramenta do Unix *size* para contar o tamanho em bytes de cada função.

A Tabela 1 apresenta os resultados para Huffman. A biblioteca de matrizes causou o acréscimo mais significativo em tamanho de programa, tanto para a versão protegida quanto para a desprotegida (27% e 37%, respectivamente). A biblioteca de Freivalds incorre em aumento de somente 8% em tamanho de programa. Note que o tamanho em bytes de cada função das bibliotecas é constante, independente do tamanho da aplicação. Assim, se a aplicação fosse maior, os tamanhos das bibliotecas seriam eficientemente amortizados. Na realidade, a biblioteca de Freivalds foi codificada com somente 246 bytes, sendo perfeitamente implementável em qualquer sistema embarcado com fortes

Tabela 1. Impacto das Operações sobre Matrizes e da Técnica de Freivalds em Tamanho de Programa para Huffman

Função		
Nome	Tamanho (bytes)	% do tamanho total[a]
Main	132	4% (6%)
Huffman Protegido	1838	61% (n/a)
Huffman Desprotegido	1249	n/a (57%)
Tamanho Total de Biblioteca		
Nome	Tamanho (bytes)	% do tamanho total[a]
Biblioteca de Matrizes	806	27% (37%)
Biblioteca de Freivalds	248	8% (n/a)
Tamanho Total de Programa		
Versão	Tamanho (bytes)	
Huffman Protegido	3024	
Huffman Desprotegido	2187	

[a]% da versão protegida (% da versão desprotegida).

restrições de recursos, seja ele crítico ou não. O incremento em tamanho de programa obedece a mesma tendência (pois as bibliotecas possuem tamanho constante independentemente do algoritmo que as utilizem) para qualquer algoritmo e é corroborado pelos resultados das transformadas MDCT/IMDCT.

6. Conclusões e Trabalhos Futuros

Neste artigo, propusemos descrever programas completamente com matrizes e realizar a correção e a detecção de falhas transitórias utilizando a técnica de Freivalds. Utilizamos Huffman e as transformadas MDCT/IMDCT completamente escritos com operações sobre matrizes, mensuramos os tempos de execução e calculamos a porcentagem das operações sobre matrizes no tempo de execução. Mostramos que as operações sobre matrizes são um componente importante nesses estudos de caso (sendo a maioria em Huffman), demonstrando a importância e a adequação da abordagem proposta para a realização de tolerância a falhas em nível de sistema.

No caso de Huffman, a implementação baseada em matrizes não é a mais comum, a qual é baseada em programação procedural sobre árvores. Essa implementação com matrizes é orientada a fluxo de dados, permitindo a aplicação da técnica de Freivalds de maneira extremamente eficiente para a proteção contra falhas transitórias. Isso é claramente benéfico, dado que a proteção de aplicações orientadas a controle utilizando a análise estática e de criticidade das variáveis é consideravelmente menos eficiente que Freivalds, como discutido nos trabalhos relacionados.

Demonstramos que o acréscimo no tempo de execução incorrido devido a Freivalds decresce com o aumento do tamanho do problema para Huffman, sendo de 5% quando $n = 256$. No caso das transformadas MDCT/IMDCT, como as operações sobre

matrizes compreendem $\sim 40\%$ do tempo total de execução, o custo de Freivalds não é tão eficientemente amortizado quanto em Huffman, mas ainda é extremamente eficiente para detecção e correção, sendo muito melhor que técnicas clássicas de tolerância a falhas como Duplicação com Comparação, fato corroborado pelos experimentos realizados. Apresentou-se que fingerprinting baseado em Freivalds supera significativamente em desempenho a DwC, demonstrando a viabilidade de implantar fingerprinting em sistemas embarcados com fortes restrições de recursos. Além disso, demonstramos que a proteção por Freivalds e as operações sobre matrizes demandam um acréscimo ínfimo em memória de programa, sendo de 248 e 806 bytes, respectivamente, independente da aplicação. Esses resultados mostram que essas técnicas podem ser facilmente empregadas em sistemas embarcados com restrição de tamanho de memória de programa.

Como trabalhos futuros, estamos projetando um mecanismo de descrição de porções orientadas a controle usando-o para induzir um anel parcialmente ordenado sobre matrizes. Com essa construção algébrica, teremos tanto as porções orientadas a controle quanto as orientadas a dados protegidas contra falhas transitórias, bastando aplicar a eficiente técnica de Freivalds em ambos os casos. Por fim, estamos escolhendo algumas funções relevantes para o domínio de sistemas embarcados aeroespaciais, funções as quais usaremos para validar nossa proposta através de um estudo de caso de grande porte.

Referências

Argyrides, C., Lisboa, C. A. L., Pradhan, D. K., and Carro, L. (2009). A fast error correction technique for matrix multiplication algorithms. In *IOLTS '09: Proc. of the 15th IEEE International On-Line Testing Symposium*, pages 133–137, Los Alamitos, CA, USA. IEEE.

Atallah, M. J., Kosaraju, S. R., Larmore, L. L., Miller, G. L., and Teng, S.-H. (1989). Constructing trees in parallel. In *SPAA '89: Proc. of the 1st Ann. ACM Symposium on Parallel Algorithms and Architectures*, pages 421–431, New York, NY, USA. ACM.

Blum, L., Shube, M., and Smale, S. (1989). On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines. *Bulletin (New Series) of the American Mathematical Society*, 21(1):1–46.

Blum, M. and Kanna, S. (1989). Designing programs that check their work. In *STOC '89: Proc. of the 21st Annual ACM Symposium on Theory of Computing*, pages 86–97, New York, NY, USA. ACM.

Chen, G. and Kandemir, M. (2005). Improving java virtual machine reliability for memory-constrained embedded systems. In *DAC '05: Proc. of the 42nd Annual Design Automation Conference*, pages 690–695, New York, NY, USA. ACM.

Cheng, M.-H. and Hsu, Y.-H. (2003). Fast imdct and mdct algorithms - a matrix approach. *IEEE Transactions on Signal Processing*, 51(1):221–229.

Cheynet, P., Nicolescu, B., Velazco, R., Rebaudengo, M., Sonza Reorda, M., and Violante, M. (2000). Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors. *IEEE Transactions on Nuclear Science*, 47(6):2231–2236.

Coppersmith, D. and Winograd, S. (1987). Matrix multiplication via arithmetic progressions. In *STOC '87: Proc. of the 19th Annual ACM Symposium on Theory of Computing*, pages 1–6, New York, NY, USA. ACM.

Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., and Xiao, C. (2007). The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45.

Florio, V. D. and Blondia, C. (2008). A survey of linguistic structures for application-level fault tolerance. *ACM Computing Surveys*, 40(2):1–37.

Huang, K.-H. and Abraham, J. A. (1984). Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, 33(6):518–528.

Huffman, D. A. (1952). A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101.

Krishnamurthi, S. and Reiss, S. P. (2003). Automated fault localization using potential invariants. In *AADEBUG '03: Proceedings of the International Workshop on Automated and Algorithmic Debugging*, pages 1–4.

Lisboa, C., Erigson, M., and Carro, L. (2007). System level approaches for mitigation of long duration transient faults in future technologies. In *ETS '07: Proc. of the 12th IEEE European Test Symposium*, pages 165–172, Los Alamitos, CA, USA. IEEE.

Lisboa, C. A., Grando, C. N., Moreira, A. F., and Carro, L. (2009). Building robust software using invariant checkers to detect run-time transient errors. In *DFR '09: Proceedings of the 1st Workshop on Design for Reliability*, pages 48–54.

Motwani, R. and Raghavan, P. (1995). *Randomized algorithms*. Cambridge University Press, New York, NY, USA.

Oh, N., Mitra, S., and McCluskey, E. J. (2002). Ed4i: Error detection by diverse data and duplicated instructions. *IEEE Transactions on Computers*, 51(2):180–199.

Pattabiraman, K., Kalbarczyk, Z., and Iyer, R. K. (2007). Automated derivation of application-aware error detectors using static analysis. In *IOLTS '07: Proceedings of the 13th IEEE International On-Line Testing Symposium*, pages 211–216, Washington, DC, USA. IEEE.

Prata, P. and Silva, J. G. (1999). Algorithm based fault tolerance versus result-checking for matrix computations. In *FCTS '99: Proc. of the 29th Annual International Symposium on Fault-Tolerant Computing*, pages 4–11, Los Alamitos, CA, USA. IEEE.

Princen, J., Johnson, A., and Bradley, A. (1987). Subband/transform coding using filter bank designs based on time domain aliasing cancellation. In *ICASSP '87: Proc. of the Int. Conf. on Acoustics, Speech, and Signal Processing*, volume 12, pages 2161–2164, Los Alamitos, CA, USA. IEEE.

Vemu, R., Gurumurthy, S., and Abraham, J. (2007). ACCE: Automatic correction of control-flow errors. In *ITC '07. IEEE International Test Conference*, pages 1–10.

XI Workshop de Testes e Tolerância a Falhas

♦

Sessão Técnica 4

Serviços Web e Roteamento

Ampliando a Disponibilidade e Confiabilidade em Ambientes de Serviços Web *Stateful*

Igor Nogueira Santos[1], **Daniela Barreiro Claro**[1], **Marcelo Luz**[1]

[1]Laboratório de Sistemas Distribuídos (LaSiD)
Departamento de Ciência da Computação
Instituto de Matemática / Universidade Federal da Bahia (UFBA)
Av. Adhemar de Barros, s/n, Ondina – Salvador – BA – Brasil

igorns@gmail.com, dclaro@ufba.br, celoluz@gmail.com

Abstract. *Web services are being widely used in applications which require high availability and reliability. Several specifications have been created in order to standardize the use of reliable mechanisms on Web services. Web services have been replicated willing to improve its availability. Considering that WS are autonomous and heterogeneous, their replication is even harder when there is state maintenance because web services are developed by different organizations in different ways. This paper evaluates some related work and introduces an hybrid and a passive replication mechanism with state maintenance on web services. Such approach was evaluated throw a local network so as to analyze the overhead obtained and was developed for Axis2 because this engine is widely used for development of web services. Our results presented a satisfactory performance in order to guarantee the replication in stateful web services.*

Resumo. *Os serviços web estão sendo cada vez mais utilizados em aplicações que demandam alta disponibilidade e confiabilidade. Diversas especificações têm sido criadas com o intuito de padronizar a utilização de mecanismos confiáveis para serviços web. Serviços web têm sido replicados, ampliando, consequentemente, a sua disponibilidade. Considerando que os serviços web são autônomos e heterogêneos além de manter o estado (stateful web service), a replicação de serviços web é uma tarefa árdua e complexa, visto que diferentes empresas podem publicar seus serviços em maneiras distintas. O presente trabalho propõe avaliar trabalhos relacionados e introduzir um mecanismo de replicação passiva e híbrida com a manutenção de estados. Este mecanismo foi avaliado em uma rede local com o intuito de analisar o overhead causado, além de ter sido desenvolvido para o ambiente Axis2, visto ser atualmente o ambiente mais utilizado para o desenvolvimento de serviços web. Os resultados apresentaram um desempenho satisfatório em relação à garantia de replicação de serviços web stateful.*

1. Introdução

A crescente utilização de sistemas computacionais leva a uma necessidade de tolerar falhas, especialmente em sistemas críticos. Estes sistemas têm sido, cada vez mais, incrementados com o intuito de serem confiáveis o suficiente para os usuários. A disseminação

das redes, principalmente a Internet, tem aumentado o número de sistemas críticos distribuídos. Alguns destes sistemas utilizam o formato de serviços *web* (Web Services-WS) para publicar as suas funcionalidades, principalmente porque os WS utilizam tecnologias padronizadas (SOAP e WSDL). Além disso, eles são autônomos e heterogêneos, ou seja, são desenvolvidos e publicados independentemente de linguagens, protocolos ou interfaces.

Nos últimos anos, algumas especificações foram desenvolvidas com o intuito de aprimorar as características de confiabilidade nos serviços *web*, ampliando, assim, a sua utilização nestes sistemas críticos. Segurança (WS-Security [Lawrence and Kaler 2004]) e mensagens confiáveis (WS-ReliableMessaging [Iwasa et al. 2004]) são exemplos destas especificações. Alta disponibilidade, um importante critério em sistemas confiáveis (*dependable systems* [Avizienis et al. 2004]), é um dos aspectos que ainda não foram tratados por nenhuma especificação. Um meio de aprimorar a disponibilidade em sistemas distribuídos é replicando cada componente em diferentes servidores. Assim, se um componente falha, outro pode substituí-lo. Especificamente para WS, esta abordagem possui algumas dificuldades devido à heterogeneidade das plataformas onde os mesmos são publicados, impedindo, consequentemente, a determinação de um protocolo de replicação. Além disso, caso o estado de um serviço *web* seja considerado (*stateful web* services), um processo de sincronização de estados deve ser utilizado.

Com o intuito de sobrepor estas limitações, alguns *middlewares* de replicação têm sido propostos, tais como: FT-SOAP [Chen 2007], WS-Replication [Jiménez-Peris et al. 2006], Conectores Tolerantes a Falhas [Fabre and Salatge 2007] e replicação híbrida [Froihofer et al. 2007]. O presente trabalho apresenta um mecanismo transparente de replicação passiva e híbrida para serviços *web stateful* desenvolvidos através do ambiente Axis2 [Axis2 2010]. Este mecanismo garante que os estados dos serviços sejam mantidos, mesmo na presença de falhas. Diferente dos mecanismos propostos anteriormente que implementam replicação passiva ou híbrida e requerem uma modificação específica no WS desenvolvido, nenhuma modificação no WS é requerida nesta proposta. Além disso, este trabalho reduz a computação em cada réplica, visto que ele processa requisições somente na presença de defeitos.

O presente trabalho está organizado como se segue: a seção 2 apresenta alguns aspectos de tolerância a falhas. Na seção 3, são apresentados os trabalhos relacionados com o intuito de melhor posicionar o presente trabalho. A seção 4 apresenta a implementação proposta e os algoritmos desenvolvidos. A seção 5 apresenta os experimentos realizados e os resultados. Seção 6 apresenta as conclusões e os trabalhos futuros.

2. Aspectos de Tolerância a falhas e Serviços *web*

Confiabilidade (*dependability*) é um critério de qualidade composto por outros critérios tais com Integridade, Manutenibilidade e Disponibilidade [Avizienis et al. 2004]. Estes critérios se tornam mais importantes quando as interações com os WS ficam automáticas. Mais ainda, um defeito pode não somente afetar um único WS, mas uma composição deles. Assim, técnicas de tolerância a falhas têm sido amplamente utilizadas, visto que elas podem garantir a continuidade da execução de um serviço devido ao seu mecanismo de redundância, mesmo na presença de falhas.

Alguns mecanismos de replicação podem ser utilizados para tolerar falhas usando

um conjunto especial de características. Estas características determinam quão adaptado estes mecanismos estão quando são aplicados em um contexto específico. Um contexto específico, por exemplo, um sistema distribuído, é basicamente suportado por processos e comunicações [Cristian 1991]. Técnicas de tolerância a falhas, quando aplicadas a este contexto (sistema distribuído), implementam a sua redundância através do hardware, por exemplo, replicando os servidores em diferentes locais e, consequentemente, os seus *softwares*. Entre os principais modelos de falha que um sistema distribuído pode lidar, destacam-se os seguintes:

1. Falhas Bizantinas: servidores podem ter um comportamento malicioso, provavelmente se unindo a outros servidores falhos;
2. Falhas por Omissão: se um servidor perder mensagens mesmo que o risco de dano seja mínimo, então este servidor pode falhar por omissão;
3. Falhas por Parada (*Crash*): se depois de uma primeira omissão o servidor parar de mandar ou receber mensagens até ele ser reinicializado, então houve uma parada silenciosa neste servidor;
4. Falhas Temporais: ocorre quando uma requisição que deveria receber uma resposta dentro de um intervalo de tempo tem um atraso. Estas falhas ocorrem em sistemas síncronos, quando um tempo máximo é pré-estabelecido. No caso do servidor ultrapassar este limite de tempo, então este servidor é suspeito de falha.

   Neste trabalho, somente as falhas por parada silenciosas (*crash*) são tratadas.

Um mecanismo de replicação requer a manutenção de um estado consistente entre as réplicas. Isso pode assegurar que na presença de uma falha no servidor, o serviço pode continuar sua execução utilizando outra réplica a partir do ponto onde ocorreu a falha. Comunicação em grupo representa um dos principais meios de se construir um sistema distribuído replicado. Neste sentido, um conjunto de processos têm suas atividades coordenadas com o intuito de manter um estado consistente quando suas funções são executadas. Detecção de falhas, entrega *multicast* e ordem das mensagens são características fundamentais de comunicação em grupo e são muito utilizadas no desenvolvimento de mecanismos (*middlewares*) de replicação.

A ordenação das mensagens representa a manipulação de todas as requisições realizadas para um grupo e é essencial para manter o estado consistente dentro de cada réplica. Existem dois tipos básicos de ordenação:

1. Ordenação FIFO: entrega das mensagens para todos os membros segundo ordenação FIFO;
2. Ordenação Total: garante que todos os membros recebem todas as mensagens na mesma ordem, assim que as mensgens são recebidas durante o tempo.

O mecanismo de replicação utiliza um protocolo para garantir a sincronização do estado entre as réplicas. O protocolo de replicação pode ser dividido em dois grupos: protocolo com o processamento moderado e protocolo com o processamento redundante [Défago and Schiper 2001].

2.1. Processamento Moderado (*Parsimonious Processing*)

O principal protocolo que aplica o processamento moderado é a replicação passiva. Nesse protocolo, todas as requisições dos clientes são processadas por uma única réplica - a

réplica primária. As outras réplicas, chamadas *backup*, recebem somente atualizações de estado do servidor primário. Dessa forma, a ordenação FIFO de mensagens é suficiente para garantir a consistência entre as réplicas, já que somente as mensagens do membro primário são ordenadas.

Na falha da réplica primária, outra deve assumir seu lugar. Porém, durante o intervalo de tempo específico no qual o novo membro primário estiver sendo eleito, as novas requisições dos clientes serão perdidas, já que não existirá um servidor primário para processá-las. Nesse sentido, a replicação passiva clássica não é totalmente transparente ao usuário, de forma que o mesmo poderá necessitar re-enviar a mensagem, caso nenhuma resposta seja retornada. Outras abordagens de processamento moderado estão sendo utilizadas para contornar esta limitação de maneira eficiente, dentre elas, destacam-se a *coodination-cohort* e a replicação semi-passiva [Défago and Schiper 2001].

2.2. Processamento Redundante (*Redundant Processing*)

No processamento redundante, todas as requisições são processadas por todas as réplicas de forma que seja garantido um tempo constante de resposta, ainda que na presença de falhas. O principal protocolo desse grupo é a replicação ativa.

Na replicação ativa, todos os gerenciadores de réplicas agem como máquinas de estados que desempenham as mesmas atividades e que estão organizadas em grupos. Uma máquina de estados é formada por variáveis que encapsulam as informações de seu estado e de comandos que modificam esse estado ou produzem uma resposta [Schneider 1990]. Nesse modelo, cada comando consiste em um programa determinístico cuja execução é atômica em relação a outros comandos, de modo que a execução de uma máquina é equivalente a efetuar operações em uma ordem estrita. Dessa forma, o estado de uma máquina é uma função determinística de seus estados iniciais e da sequência de comandos neles aplicados.

A utilização desse modelo como protocolo de replicação só é possível se cada réplica começar no mesmo estado inicial e executar a mesma série de comandos, na mesma ordem, de forma que cada máquina produzirá os mesmos resultados para entradas iguais. Assim, a replicação ativa requer que o processamento de mensagens seja determinístico, o que impede, por exemplo, a utilização desse mecanismo em aplicações *multithreading*.

Na replicação ativa, todas as réplicas podem ser diretamente invocadas pelos usuários, portanto, as mensagens devem ser ordenadas em relação a todas as réplicas, à medida em que são recebidas. A ordenação total garante essas características.

Com a crescente utilização de WS em sistemas que demandam alta confiabilidade, a aplicação das técnicas de tolerância a falhas nesses componentes é cada vez mais comum. O presente trabalho visa incorporar no Axis2 um mecanismo de replicação passiva e híbrida transparente para serviços *web stateful*.

2.3. Processamento Híbrido

Além dos esquemas vistos, é possível construir novas abordagens que combinam elementos de processamento moderado com elementos de processamento redundante. O modelo de replicação aplicado em [Froihofer et al. 2007] é um exemplo deste aspecto.

3. Trabalhos relacionados

O requisito de disponibilidade, um dos principais componentes da confiabilidade de um sistema, é particularmente difícil de atingir no ambiente heterogêneo em que serviços *web* habitualmente operam. A tarefa de construir modelos de falha, por exemplo, é dificultada pela natureza da publicação dos serviços, já que a descrição de um serviço, obrigatoriamente, especifica apenas informações básicas necessárias à sua invocação, de modo que informações sobre aspectos não-funcionais, tais como disponibilidade e desempenho, não são publicadas. Dessa forma, determinar quais as possíveis falhas que um determinado serviço pode vir a apresentar pode ser impossível sem as informações necessárias sobre cada serviço que o compõe. De fato, a disponibilidade de um serviço pode ser até menor que qualquer um dos componentes que lhe dão forma [Moser et al. 2007].

Apesar da evidente necessidade, nenhuma especificação de confiabilidade no que tange à disponibilidade de serviços ainda foi desenvolvida. Isso ocorre, em grande parte, devido à dificuldade de estender as técnicas de replicação além das fronteiras de uma única organização, já que as tecnologias que elas usam no desenvolvimento de seus serviços são, possivelmente, não equivalentes. Nos últimos anos, vários *middleware* de replicação em serviços *web* foram desenvolvidos com o propósito de contornar essa limitação, dentre eles FT-SOAP [Chen 2007], WS-Replication [Jiménez-Peris et al. 2006], Conectores de Tolerância a Falhas [Fabre and Salatge 2007] e a replicação híbrida adotada em [Froihofer et al. 2007].

3.1. FT-SOAP

FT-SOAP [Chen 2007] é um mecanismo de replicação passiva. Os componentes básicos dessa especificação são: o **Gerenciamento de Falhas** que é utilizado para realizar o monitoramento de estado das réplicas, o **Mecanismo de Log e Recuperação**, responsável por fazer o *log* das invocações, de forma que elas não sejam perdidas na falha do gerenciador primário, e o **Gerenciador de Replicação**, que tem a função de monitorar e constituir os grupos de réplicas.

O uso de componentes centralizados adiciona novos pontos de falhas ao sistema, de forma que até os componentes do próprio *middleware* devem ser replicados.

No mecanismo proposto neste artigo, cada réplica funciona como um mecanismo de replicação independente, evitando a centralização encontrada na proposta FT-SOAP.

3.2. WS-Replication

WS-Replication [Jiménez-Peris et al. 2006] implementa replicação ativa no contexto dos serviços *web*. O protocolo de replicação aplicado é baseado no componente WS-Multicast que utiliza tecnologias básicas de WS (SOAP e WSDL) para promover sincronia entre as réplicas. Ao utilizar o WS-Multicast em sua estrutura, WS-Replication mantém na estrutura de replicação a independência de plataformas inerente à arquitetura SOA, escalando o protocolo a operar diretamente sobre os serviços na Internet. Além disso, como nenhuma modificação é imposta à implementação dos serviços, a replicação é totalmente transparente aos usuários.

O mecanismo de replicação implementado neste artigo, de maneira semelhante ao WS-Replication, é transparente aos usuários. No entanto, ao aplicar a replicação passiva, o mecanismo proposto requer menor processamento.

3.3. Conectores

Este mecanismo [Fabre and Salatge 2007] parte do princípio de que criar serviços *web* confiáveis é difícil porque eles geralmente dependem de outros serviços que não são confiáveis, de forma que é necessário encontrar meios externos ao serviço para prover mecanismos adicionais de tolerância a falhas. A idéia básica dessa infraestrutura é realizar a comunicação com os serviços *web* através de conectores que implementam características de replicação.

A noção de conector é baseada no conceito de ADLs (*Architecture Description Language*), que permite customizar interações entre componentes. Um conector específico de tolerância a falhas (*Specific Fault Tolerant Connector*) é utilizado para interceptar invocações a um WS e realizar ações características de mecanismos de replicação, funcionando como um componente intermediário entre um serviço *web* construído de forma não-confiável e um cliente que representa uma aplicação SOA crítica (*Critical SOA Based application*).

A presença de conectores em cada réplica de um serviço torna esse *middleware* bastante flexível, de maneira que é possível configurá-lo para aplicar replicação passiva ou ativa, além de possibilitar a utilização de réplicas não idênticas de uma mesma funcionalidade. Na replicação passiva, porém, para que haja manutenção de sincronia de estado entre as réplicas, os serviços devem implementar funções de manipulação de estado e publicá-las conjuntamente às funcionalidades propriamente ditas.

Diferentemente da proposta apresentada em [Fabre and Salatge 2007], no mecanismo desenvolvido e acoplado à plataforma Axis2, nenhuma alteração é imposta à codificação dos serviços replicados.

3.4. Modelo Híbrido de Replicação

Esse *middleware* [Froihofer et al. 2007] de replicação considera serviços publicados sobre infraestruturas homogêneas, de forma que é possível adotar medidas de replicação comumente utilizadas em objetos distribuídos. O modelo de replicação adotado tem o seu funcionamento básico baseado nos protocolos de replicação passiva e ativa.

Da passiva, ele herda a característica de todas as mensagens serem enviadas somente ao gerenciador primário. Da ativa, ele implementa o processamento redundante. O esquema de funcionamento pode ser observado na figura 1.

A mensagem chega à interface de transporte do gerenciador primário (passo 1), segue o fluxo de entrada até ser interceptada (passo 2). Nesse momento, a invocação é serializada e enviada a todas as réplicas através do *toolkit* SPREAD (passo 3). Ao receber a invocação, o interceptor a remonta (passo 4) e a envia para o início do fluxo de entrada (passo 5), de forma que a réplica processe a mesma invocação. Para garantir a sincronia, a réplica primária interrompe o fluxo de processamento (passo 2) e só o reinicia depois que a invocação é recebida novamente, uma vez que a réplica primária, necessariamente, também executa os passos 4 e 5.

O mecanismo de interceptação deste último *middleware* é implementado na *engine* do Axis, de forma que o protocolo de replicação não precisa ser implementado nos serviços em si. O mecanismo, porém, só é aplicado sobre serviços publicados através

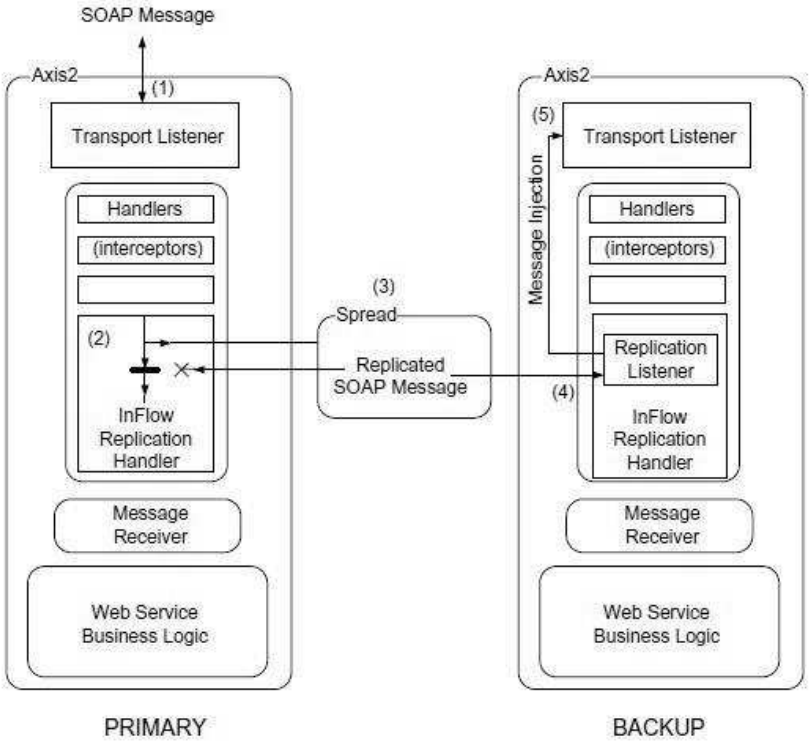


Figura 1. Arquitetura da Replicação Híbrida [Froihofer et al. 2007].

dessa ferramenta. Apesar desta limitação, a plataforma Axis é uma das principais formas de publicação de serviços *web* da atualidade, o que facilita a utilização deste mecanismo.

O mecanismo de interceptação e replicação deste *middleware* serviu de base para o desenvolvimento da proposta do presente trabalho. O principal diferencial foi a incorporação da replicação passiva clássica para serviços e, com isso, foi obtida uma diminuição no processamento de cada réplica. Na proposta implementada, porém, ao momento da interceptação da invocação e *multicast* da informação, não somente a invocação é enviada às demais réplicas, mas todo o contexto da invocação, de forma que ao receber esse contexto, as réplicas backup não necessitam re-inserir a invocação no início dos fluxos de entrada, mas somente dar prosseguimento ao processo do ponto de onde ele parou na réplica primária. A implementação desta replicação passiva é detalhada na próxima seção.

4. Implementação

O mecanismo proposto neste trabalho foi implementado para a versão mais nova do Axis, *engine* Axis2, com o intuito de garantir que nenhuma modificação no serviço *web* precisa ser realizada. Dois modelos de replicação (passiva e híbrida) foram implementados e testados para serviços *web stateful*, levando em consideração a manutenção de consistência de estado entre as réplicas. Os serviços testados foram desenvolvidos em Axis2 [Axis2 2010] e as garantias de comunicação em grupo entre as réplicas foram implementadas através da ferramenta (*toolkit*) JGroups [Ban 2008].

JGroups é um *toolkit* Java para comunicação *multicast* confiável. Sua arquitetura

consiste de três partes: uma API Canal (*Channel*) que provê as funcionalidades básicas de acesso e criação de grupos de réplicas; Blocos de Implementação (*Building Blocks*) que provêm uma abstração mais refinada de utilização do canal; e Pilha de Protocolos (*Protocol Stack*) que é formada pelos componentes que implementam as garantias de confiabilidade e ordenação da entrega de mensagens.

Já o Axis2 é um *engine* de implementação da especificação SOAP. Seu funcionamento consiste, basicamente, em enviar e receber mensagens XML. Para tanto, a arquitetura do Axis2 mantém dois fluxos que executam essas atividades. Esses fluxos são implementados pelo mecanismo Axis (*Axis Engine*) através de dois métodos: enviar (*send*) e receber (*receive*). Os dois fluxos são chamados, respectivamente, Fluxo de Entrada (*InFlow*) e Fluxo de Saída (*OutFlow*).

Nesse modelo, cada fluxo é dividido em fases (*phases*) e cada fase é formada de *handlers* que agem como interceptadores processando partes da mensagem e provendo qualidade de serviço. Quando uma mensagem SOAP está sendo processada, os *handlers* registrados nas fases são executados. As informações de execução desses *handlers* são armazenadas em objetos de contexto (*context*) que têm por finalidade manter dados que podem ser compartilhados entre várias invocações ou entre *handlers* de uma única invocação como, por exemplo, a sessão. Dessa forma, para adicionar funcionalidades ao processamento, é necessário registrar novos *handlers* nas fases de execução pré-existentes na arquitetura, ou em novas fases criadas pelo usuário.

Dois modelos de replicação foram implementados e testados: replicação passiva clássica e um modelo híbrido. O esquema de interceptação foi baseado no trabalho desenvolvido em [Froihofer et al. 2007], porém foi modificado a fim de melhorar a distribuição de mensagens no grupo.

4.1. Replicação Híbrida

Uma nova fase foi adicionada aos fluxos de entrada e saída intitulada *FaseReplicacaoHibrida*. O funcionamento do protótipo, no fluxo de entrada, pode ser observado na figura 2.

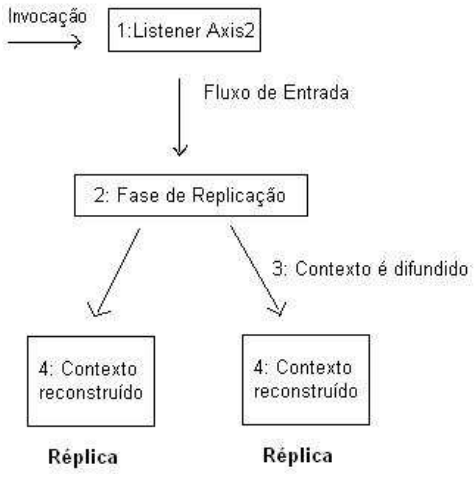


Figura 2. Funcionamento da Replicação Híbrida.

1. A invocação do cliente, interceptada pelo *listener*, chega à interface de transporte da réplica primária e é encaminhada às fases posteriores do fluxo de entrada. Durante as fases, cada *handler* recebe um objeto do tipo contexto e adiciona ou modifica informações no mesmo. Basicamente, um contexto contém informações sobre a configuração da engine Axis2, da sessão e o envelope SOAP em si;
2. Ao atingir a fase de replicação, o contexto é serializado e não somente a invocação, o que difere da solução proposta em [Froihofer et al. 2007]. Essa modificação é feita porque a fase de replicação foi inserida imediatamente antes da fase de processamento, de forma que o contexto ao qual o *handler* implementado tem acesso está completo e pronto para dar início ao processamento da requisição, tornando desnecessária a reconstrução do mesmo em cada réplica.
3. Serializado o contexto, a mensagem de atualização é enviada a todas as réplicas, exceto a réplica primária, que já possui a informação construída. Essa aproximação é possível, porque o uso do JGroups possibilita à réplica primária esperar por nenhuma, a primeira, a maioria ou todas as confirmações de recebimento das réplicas, de forma que a consistência de estados é garantida pela comunicação em grupo, dispensando o envio do contexto serializado para a mesma réplica que o gerou. O envio *multicast* é feito utilizando o bloco de implementação *RPCDispatcher* que permite a invocação remota de métodos.
4. No recebimento do contexto, as réplicas executam o algoritmo 1.

Algorithm 1 Recebimento do contexto pelas réplicas *backup*

```
1: context = unserialize(serializedContext);
2: context.activate;
3: AxisEngine.resume(context)
```

Ao ser serializado, o contexto é transformado em uma cadeia de caracteres codificados a fim de garantir a segurança da informação no ínterim em que trafega pela rede. Com o contexto reconstruído (linha 1), faz-se necessário reativá-lo para que o processamento seja reiniciado. No mecanismo implementado, o processamento da requisição é retomado de uma maneira otimizada, pois, não é necessária a reinserção do invocação no início do fluxo como faz a aproximação adotada em [Froihofer et al. 2007], de forma que a operação **AxisEngine.resume** dispara, imediatamente, a fase de processamento na réplica em questão, como pode ser observado na linha 3. Dessa forma, o processamento redundante é atingido.

Caso o gerenciador primário falhe, o novo líder é determinado através da nova lista de membros criada pelo JGroups. Para tanto, todas as réplicas têm acesso à listas de membros (*views*) idênticas, de forma que o novo líder escolhido é aquele que ocupa a primeira posição (posição zero) na *view*. Como cada réplica atualiza o seu estado ao recebimento de cada requisição, o estado é mantido e, dessa forma, o novo líder eleito está consistente para tratar as novas requisições.

Assim, o protótipo desenvolvido mantém o estado entre as réplicas, já que todas as requisições são processadas por todos os membros do grupo de replicação, apesar de somente o membro primário receber e responder às invocações dos clientes. A consistência de estado é garantida através da utilização do protocolo FIFO na pilha do JGroups.

4.2. Replicação Passiva Clássica

Semelhante ao protótipo de replicação híbrida, também foi adicionada uma nova fase ao fluxo de entrada (*FaseReplicacaoPassiva*). As etapas pelas quais a replicação passiva realiza suas atividades são bastante próximas da replicação híbrida, exceto que as réplicas agem de forma distinta ao recebimento do contexto serializado. Os passos que a replicação passiva aplica no fluxo de entrada são os seguintes:

1. Antes da fase de processamento, o contexto é serializado e enviado às réplicas de forma semelhante aos passos 2 e 3 do fluxo de entrada no esquema híbrido. Como o processamento é moderado, nenhum mecanismo de registro de espera por confirmação de execução é necessário. Além do contexto em si, o nome do serviço que está sendo invocado também é enviado;
2. No recebimento do contexto, as réplicas executam os passos enumerados no algoritmo abaixo:

Algorithm 2 Recebimento do contexto pelas réplicas *backup*

```
1: serviceName = context.getServiceName
2: contextState = contexto.getSerializedValue;
3: historic.save(serviceName, contextState);
```

A mensagem de atualização para a réplica contém dois campos, sendo o primeiro deles o nome do serviço e o segundo o contexto serializado. Na linha 3, o contexto serializado é armazenado em um *hashmap* mantido em cada réplica. É possível notar que, a cada requisição a um determinado serviço, seu histórico é sobrescrito com o novo contexto. Isto é pertinente porque as informações de sessão são mantidas no contexto do Axis2, de forma que o contexto mantém as modificações feitas aos estados da sessão. Para suportar persistência de estado em dispositivos secundários como arquivos ou bancos de dados, o histórico pode ser facilmente estendido de forma a armazenar uma lista de contextos em cada serviço. Cada contexto corresponderia a uma invocação e assim, na recuperação de estado, todas as entradas do histórico teriam que ser reconstruídas e processadas na nova réplica primária antes que a nova requisição fosse processada.

Dessa forma, em cada réplica, antes da execução do fluxo acima, o histórico é verificado à procura de entradas pelo nome do serviço. Como a réplica primária nunca recebe as mensagens de atualização que envia, necessariamente, o seu histórico sempre estará vazio. Na ocorrência de falhas, quando o novo primário é eleito e recebe uma invocação, ele busca no histórico algum contexto associado ao nome do serviço. Caso encontre, esse contexto é reconstruído e processado antes que a nova requisição do usuário seja atendida.

Por utilizar o processamento moderado, o tempo de recuperação de estado é proporcional ao número de requisições que devem ser processadas antes da execução da nova invocação. Se for considerado o contexto do Axis2, então esse tempo é fixo e igual a uma requisição a mais. Caso o histórico seja modificado a fim de manter todo o conjunto de requisições, então esse valor pode ser bem maior dependendo do intervalo de tempo decorrido entre a falha do servidor primário e o início do recebimento das invocações.

A ordenação FIFO de mensagens é suficiente para manter a consistência de estado entre as réplicas. O estado é mantido através do *hashing* de contextos replicado

em cada réplica. Na falha do líder, o novo gerenciador primário irá recuperar o estado reaplicando as requisições feitas a cada serviço, isoladamente, de forma que o mínimo de processamento seja feito na fase de recuperação. Dessa forma, cada membro do grupo de replicação funciona como um mecanismo de replicação em si, dispensando a criação de componentes centralizados e a conseqüente adição de novos pontos de falha ao modelo.

5. Experimentos e resultados

Com o intuito de avaliar o funcionamento dos protótipos desenvolvidos foram realizados testes de desempenho para determinar o *overhead* da utilização do mecanismo no que tange ao trabalho implementado e a utilização da comunicação em grupo. Também foram validadas a manutenção e consistência de estado entre as réplicas.

5.1. Configuração do Ambiente

Os experimentos iniciais foram realizados de forma a considerar somente o tempo de execução dos protótipos sem influência da latência de redes. Para tanto, foram criadas quatro instâncias independentes do servidor Tomcat, versão 6.0.18, em um PC Pentium D 2.80 GHz, 1.5 GB de RAM, *Microsoft Windows XP Professional* 2002 *Service Pack 2*. Cada instância Tomcat publica uma instância Axis2.1.4.1. Já a versão do JGroups toolkit é a 2.6.4 e a pilha padrão de protocolos foi utilizada.

Para a realização dos testes foi desenvolvido um serviço *web* simples, que mantém um estado (*stateful*). As operações básicas oferecidas pelo serviço estão listadas a seguir:

1. Criar sessão: Método responsável por criar um vetor na sessão;
2. Adicionar elemento: Adiciona um elemento ao vetor na sessão;
3. Listar número de elementos: Retorna a quantidade de itens no vetor;
4. Listar elementos: Retorna os elementos na sessão em uma string.

Nos experimentos, cada requisição adicionou um elemento ao vetor da forma ”elemento i”, onde i cresce de acordo com o número da requisição, variando, dessa forma, de ”elemento 0”a ”elemento 2999”.

Esse serviço foi publicado em cada instância Axis2 instalada no computador. Para o teste de desempenho, foi feita uma série de 3000 requisições à réplica primária e calculados a média e o desvio padrão para os seguintes cenários:

1. Um serviço, sem qualquer adição de replicação;
2. Replicação com 1, 2, 3 e 4 réplicas.

5.2. Conjunto de testes - Número de réplicas

Para avaliar o overhead dos mecanismos em si, foram aplicados testes nos protótipos na presença de uma única réplica, de forma a evitar o overhead da comunicação em grupo. Os valores obtidos estão de acordo com o gráfico da figura 3.

Os testes foram executados 5 vezes a fim de determinar o intervalo de confiança do experimento. Para um nível de 95%, a média de execução para o cenário sem replicação, foi de 46,77 milissegundos com um desvio padrão de 0,11 millesegundos. A média para a replicação passiva foi de 50,49 milissegundos para o intervalo de 0,44 milissegundos, impondo um *overhead* de 7,9%. Já o valor da replicação híbrida, teve média 55,79 milissegundos para o intervalo de 0,27 aumentando o tempo médio em 19,2%.

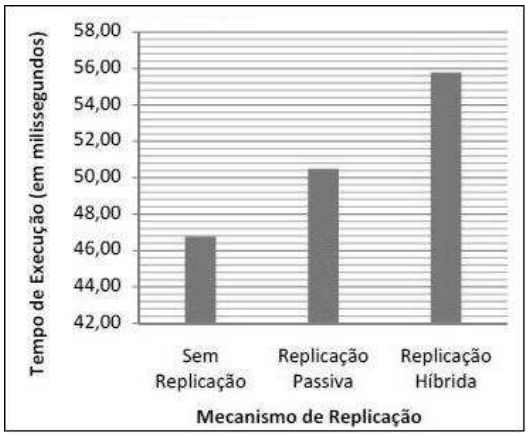


Figura 3. Gráfico comparativo dos protótipos de replicação.

O aumento no tempo de execução imposto pelo mecanismo é justificável porque a adição de uma fase representa um aumento na execução geral de mensagens, pois cada fase adicionada ao fluxo representa mais um passo de execução na *engine* do Axis2. Outro aspecto importante é a serialização do contexto. À medida que as invocações são feitas e mais informações são adicionadas à sessão, o contexto aumenta de tamanho de forma que a sua serialização representa um dos principais aspectos de consumo de tempo no modelo proposto e explica o desvio padrão relativamente alto, já que as últimas requisições da série demandaram um maior esforço de serialização em relação às requisições iniciais.

Testes foram realizados na presença de mais de uma réplica para que o overhead da comunicação em grupo pudesse ser determinado. Assim, com a adição de réplicas, o mecanismo de comunicação em grupo é efetivamente utilizado, de forma que o tempo de execução de cada mensagem, além dos fatores considerados anteriormente, é acrescido pelo envio das mensagens de atualização de estado para cada réplica, além do mecanismo de monitoração de estado dos membros do grupo, que cresce conjuntamente ao número de cópias. Nesse modelo, o *overhead* máximo foi atingido na presença de 4 réplicas, ou seja, 1 réplica primária e 3 réplicas backups. Para esse cenário com 4 cópias, o tempo médio foi de 68,70 $\pm$ 5,91 milissegundos, o que implica em um aumento de 14,5% em relação à replicação passiva com o mesmo número de réplicas e 46,9% em relação ao modelo sem replicação.

Por último, foram realizados testes em uma rede LAN, de forma que um teste mais próximo de um ambiente real fosse mensurado. O resultado dos testes pode ser observado na figura 4.

A média para a replicação passiva com duas réplicas foi 267,09 milissegundos, enquanto que o resultado para três réplicas foi 274,14 milissegundos , o que representa um aumento de 2,64% entre as configurações. Já a replicação híbrida teve média de 270,56 milissegundos para duas réplicas e 280,26 milissegundos para três. A diferença, nesse contexto, foi de 3,59%,demonstrando que o tempo de execução dos protótipos não aumenta bruscamente, em rede, com a adição de réplicas.

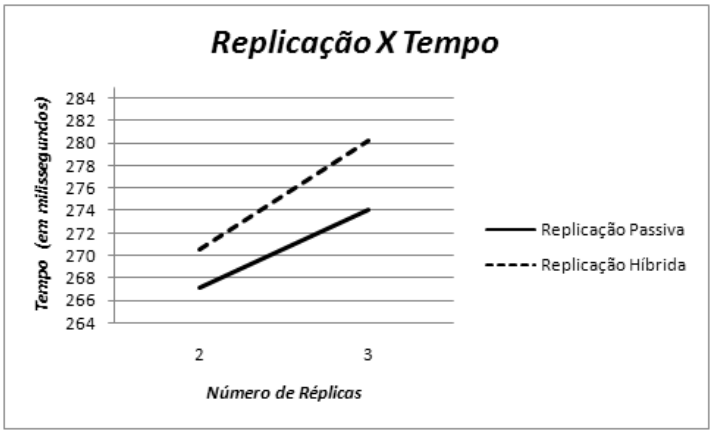


Figura 4. Resultado dos testes em rede.

5.3. Conjunto de Testes - Manutenção do Estado e Consistência

Para o teste de manutenção de estado, a análise foi feita fazendo a réplica primária falhar continuamente e aplicando a operação de listar o número de itens no novo primário eleito. Para o teste de consistência, a listagem de elementos foi invocada em cada novo primário e armazenada para posterior comparação.

O teste de manutenção de estado funcionou dentro do esperado, de forma que cada réplica apresentou 3000 itens em seus vetores após a falha do servidor primário.

O teste de consistência também foi bem sucedido. Para comparar os estados, após as invocações, a operação de listagem de conteúdo foi invocada em cada réplica e seus resultados armazenados para que pudessem ser comparados através da API Java.

Os testes demonstraram que o maior *overhead* da replicação foi a comunicação em grupo. A pilha de protocolo utilizada nos testes foi a pilha padrão do JGroups, de forma que é necessário estudá-la melhor e customizá-la a fim de apresentar um desempenho mais satisfatório.

Como todas as modificações foram feitas na arquitetura Axis2, nenhuma mudança foi necessária diretamente no serviço *web*, de forma que, nesse aspecto, este trabalho garante que serviços *web* autônomos e heterogêneos sejam replicados de uma maneira transparente.

6. Conclusão e Trabalhos Futuros

Este trabalho propôs analisar a maneira pela qual os métodos de replicação são adotados em conjunto aos serviços web. Os protótipos implementados demonstraram que a replicação em serviços é praticável, inclusive no que se refere à manutenção e consistência de estado entre as réplicas. A principal fonte de latência encontrada foi a comunicação em grupo, demonstrando que a primitiva *multicast* e o controle de membros de grupo podem ser bastante dispendiosos.

Dentre os trabalhos futuros, destaca-se uma melhor análise da pilha de protocolos do JGroups para obter um melhor desempenho, além de analisar a escalabilidade da solução proposta. Outrossim é a implementação de outros protocolos de replicação, tais como replicação semi-passiva e ativa.

Referências

Avizienis, A., Landwher, C., and Laprie, J.-C. (2004). Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing.

Axis2 (2010). Apache axis2. Disponível em: http://ws.apache.org/axis2/. Último Acesso em 31 de março de 2010.

Ban, B. (2008). Reliable multicasting with the jgroups toolkit. [S.l].

Chen, Chyouhaw; Fang, C.-L. L. D. (2007). Ft-soap: A fault-tolerant web service. Journal of Systems Architecture: the EUROMICRO Journal.

Cristian, F. (1991). Understanding fault tolerant distributed systems. Communication of ACM.

Défago, X. and Schiper, A. (2001). Specification of replication techniques, semi-passive replication, and lazy consensus. Academic Press.

Fabre, J.-C. and Salatge, N. (2007). Fault tolerance connectors for unreliable web services. 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks.

Froihofer, L., Goeschka, K., Osrael, J., and Weghofer, M. (2007). Axis2-based replication middleware for web services. IEEE International Conference on Web Services (ICWS 2007).

Iwasa, K., Durand, J., Rutt, T., Peel, M., Kunisetty, S., and Bunting, D. (2004). Web services reliable messaging tc ws-reliability 1.1. OASIS Open 2003-2004.

Jiménez-Peris, R., Patino-Martinez, M., Pérez-Sorrosal, F., and Salas, J. (2006). Ws-replication: A framework for highly available web services. WWW 2006 - International World Wide Web Conference, Edinburgh,Scotland.

Lawrence, K. and Kaler, C. (2004). Web services security: Soap message security 1.1. http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-pr-SOAPMessageSecurity-01.htm.Último acesso em: 03 de março de 2009.

Moser, L. E., Smith, P., and Zhao, W. (2007). Building dependable and secure web services. JOURNAL OF SOFTWARE, VOL. 2, NO. 1.

Schneider, F. B. (1990). Replication management using the state-machine approach. ACM Press/Addison-Wesley Publishing Co.

Multi-Priority Alternative Journey and Routing Protocol: um Algoritmo para Roteamento em Redes Tolerantes a Atrasos e Desconexões Previsíveis

Gabriel Argolo[1], Lúcia M. A. Drummond[1], Anna Dolejsi[1], Anand Subramanian[1],

[1]Instituto de Computação - Universidade Federal Fluminense
Rua Passo da Pátria 156 - Bloco E - 3º andar, São Domingos, CEP: 24210-240, Niterói - RJ

{grocha,lucia,annads,anand}@ic.uff.br

Abstract. *This work proposes the distributed algorithm Multi-Priority Alternative Journey and Routing Protocol (MP-AJRP) for routing in Delay-disruption Tolerant Networks, considering constraints such as links' bandwidth and nodes' buffer capacity. The MP-AJRP is an improved version of the AJRP algorithm and its goal is to deliver the largest number of messages to the destinations, considering alternative routes as well as multi-priorities for message scheduling. In order to evaluate the algorithm, several experiments were done using the traces produced with the contact history of the DieselNet nodes. The results showed that MP-AJRP delivered about 76% of messages and had a better performance when compared to certain algorithms that use message replication.*

Resumo. *Este trabalho propõe o algoritmo distribuído Multi-Priority Alternative Journey and Routing Protocol (MP-AJRP) para roteamento em Redes Tolerantes a Atrasos e Desconexões, considerando as restrições de largura de banda dos enlaces e de capacidade dos buffers dos nós. O MP-AJRP é uma versão aprimorada do algoritmo AJRP e tem como objetivo entregar o maior número de mensagens aos destinos considerando rotas alternativas e múltiplas prioridades para escalonamento de mensagens. Com o intuito de avaliar o algoritmo, foram realizados experimentos utilizando os traces com o histórico de contatos dos nós da rede DieselNet. Os resultados mostraram que o MP-AJRP entregou em torno de 76% das mensagens e obteve desempenho superior a determinados algoritmos que consideram a replicação de mensagens.*

1. Introdução

Redes Tolerantes a Atrasos e Desconexões (DTN - *Delay-disruption Tolerant Network*) possibilitam a transmissão de dados quando dispositivos móveis estão conectados intermitentemente. Neste tipo de rede, a comunicação entre os nós é feita diretamente ou através de nós intermediários que atuam como roteadores. A conectividade intermitente pode ser resultado da mobilidade dos nós, da potência de sinal ou até mesmo do gerenciamento de energia. DTNs são encontradas, por exemplo, em redes de sensores para monitoramento ecológico, na comunicação entre sistemas de satélites e em redes veiculares. Estas redes diferem da tradicional *Internet*, pois é assumido que esta última possui conectividade ininterrupta, além da baixa taxa de perda de pacotes e do baixo retardo de propagação.

Partindo deste pressuposto, os protocolos desenvolvidos para *Internet* cabeada são ineficazes para transmissão de dados em DTNs.

DTNs podem ser classificadas como previsíveis, também conhecidas como redes com contatos programados, imprevisíveis e probabilísticas. Na primeira, a variação da topologia ao longo do tempo é conhecida antecipadamente. As redes de satélites do tipo LEO (*Low Earth Orbit*) são um exemplo, onde as trajetórias dos satélites são previamente programadas. Na segunda, não se conhece de antemão as alterações que podem ocorrer na topologia como, por exemplo, nas redes *ad-hoc* onde os nós podem se mover arbitrariamente ao longo do tempo [Merugu et al. 2004, Oliveira e Duarte 2007]. Já nas probabilísticas, apesar de não conhecer *a priori* exatamente quando ocorrerão futuros contatos e a duração dos mesmos, é possível obter uma aproximação destes valores através da aplicação de métodos probabilísticos.

A variação da topologia da rede pode dificultar o roteamento em DTNs ao longo do tempo. A necessidade de rotear mensagens até os destinos, assim como assegurar uma comunicação eficiente, são desafios encontrados nestes tipos de rede. As desconexões frequentes causadas pelo deslocamento dos nós, o elevado tempo de permanência de mensagens nas filas e a possível inexistência de um caminho fim-a-fim são alguns dos problemas existentes.

O encaminhamento das mensagens até os destinos pode ser realizado através de diversos mecanismos como a duplicação das mensagens em nós intermediários, o encaminhamento das mensagens pelo primeiro enlace disponível ou a utilização de tabelas de roteamento [Oliveira e Duarte 2007]. Neste último caso, a construção das tabelas pode ser feita adotando-se uma abordagem centralizada, ou seja, cada nó dispõe previamente de toda informação relevante para o roteamento, ou distribuída, onde os nós não conhecem de antemão o estado global da rede. Apesar das tabelas poderem ser construídas de forma centralizada, esta abordagem apresenta inconvenientes. A necessidade de manter nos nós a informação global sobre o estado da rede torna-se difícil à medida que o tamanho da rede aumenta e também devido à intermitência dos canais de comunicação. Logo, a construção de tabelas de roteamento através de algoritmos distribuídos apresenta-se como uma solução mais apropriada.

Embora algumas estratégias de encaminhamento valham da duplicação de mensagens na rede, esta aborgadem pode acarretar em uma utilização ineficaz de recursos quando aplicado em redes com largura de banda limitada e reduzida capacidade de armazenamento dos nós. Outras formas de encaminhamento necessitam do conhecimento prévio de toda a rede, apesar disto não ser simples de obter na prática [Peleg 2000]. Além disso, até nestes casos não há garantia da entrega das mensagens, mesmo havendo uma rota fim-a-fim, dado que as mensagens podem ser perdidas devido às limitações de capacidade de armazenamento dos nós e da largura de banda dos enlaces. Portanto, o projeto de algoritmos que utilizem mecanismos de encaminhamento sem duplicação de mensagens torna-se uma alternativa interessante.

Este trabalho propõe o algoritmo distribuído *Multi-Priority Alternative Journey and Routing Protocol* (MP-AJRP) para encaminhamento de mensagens em DTNs previsíveis cujo objetivo é entregar o maior número de mensagens aos destinos, executando um mecanismo de escolha de rotas alternativas para evitar o roteamento das mensagens

por nós com *buffer* saturado. O MP-AJRP é um aprimoramento do algoritmo proposto em [Argolo et al. 2009], pois, além da abordagem *first-in-first-out* (FIFO), considera outras duas políticas de seleção de mensagens do *buffer* nos nós pertencentes às rotas das mensagens. O encaminhamento das mensagens é feito sem duplicá-las em nós intermediários e cada nó conhece apenas os intervalos de disponibilidade para comunicação com seus vizinhos e a tabela de roteamento.

As principais contribuições deste trabalho são:

- Elaboração do algoritmo MP-AJRP para encaminhamento de mensagens até os destinos considerando jornadas alternativas e múltiplas políticas de seleção de mensagens no *buffer*; e
- Realização de experimentos para avaliar o desempenho dos algoritmos propostos e comparando os resultados obtidos com outras abordagens encontradas na literatura

O restante deste trabalho está organizado da seguinte forma. A Seção 2 apresenta trabalhos relacionados ao problema de roteamento em DTNs. A Seção 3 descreve o modelo adotado para a solução do problema. A Seção 4 descreve e analisa os algoritmos distribuídos propostos. Os resultados experimentais para análise do desempenho dos algoritmos são discutidos na Seção 5. A Seção 6 apresenta as considerações finais e propostas para trabalhos futuros.

2. Trabalhos relacionados

Nesta seção são abordados diversos trabalhos encontrados na literatura com relação a propostas de algoritmos para roteamento em DTNs.

Em [Bui-Xuan et al. 2003], três algoritmos centralizados denominados *foremost journey*, *shortest journey* e *fastest journey* foram desenvolvidos com o objetivo de encontrar, respectivamente, as jornadas mais cedo, ou seja, as jornadas onde o instante de tempo de chegada da mensagem nos nós de destino é o menor possível, as jornadas com menor número de saltos e as jornadas mais rápidas, isto é, as que apresentam as menores diferenças entre o instante de tempo de chegada da mensagem no destino e o instante de envio da mesma.

O algoritmo PROPHET é proposto em [Lindgren et al. 2004] para realizar o roteamento das mensagens utilizando como base a probabilidade que os nós possuem de encontrar uns aos outros. Esta probabilidade é obtida através de cálculos realizados levando em consideração o histórico de contatos dos mesmos ao longo do tempo. Nenhuma informação quanto às alterações futuras na topologia da rede é conhecida antecipadamente. Apesar de considerar restrições na capacidade de armazenamento dos nós, o algoritmo não leva em conta as limitações na largura de banda do enlaces. Um mecanismo de replicação de mensagens é utilizado, onde as cópias das mensagens são enviadas para os nós que possuem as maiores probabilidades de encontrar os respectivos destinos.

Em [Jain et al. 2004], os autores implementam algoritmos de roteamento que, baseado em oráculos, utilizam informações sobre o estado atual e futuro da rede como os contatos entre os nós ao longo do tempo, a demanda de mensagens e a ocupação dos *buffers*. A estratégia de encaminhamento utilizada envia as mensagens para os nós intermediários até atingir os destinos sem que as mensagens sejam duplicadas. Os experi-

mentos realizados com baixa e alta carga de mensagens na rede, mostram que os algoritmos que atingem o melhor desempenho com relação ao roteamento das mesmas são os que possuem mais informações acerca das características da rede. Os algoritmos também foram avaliados variando-se a capacidade de armazenamento dos nós e a largura de banda dos enlaces.

Chen em [Chen 2005] propõe o algoritmo SGRP para roteamento em redes de satélites com trajetórias previsíveis executado em duas etapas. Inicialmente, a coleta da informação de um grupo de satélites de baixa altitude (LEOS) é feita por um satélite de médio alcance (MEO). Em seguida, os diversos MEOS trocam as informações obtidas entre si e com a informação global disponível calculam as tabelas de roteamento e as redistribuem para os LEOS.

O algoritmo Spray and Wait, elaborado em [Spyropoulos et al. 2005], considera um mecanismo de replicação que gera L cópias de cada mensagem e as distribui entre os contatos esperando que algum deles por ventura encontre o nó de destino. São analisadas algumas estratégias de priorização dos contatos que devem receber as cópias das mensagens, assim como realizada uma avaliação com relação ao número L de cópias a serem geradas considerando o tamanho da rede e a demanda de mensagens. O algoritmo não necessita de nenhuma informação prévia sobre a topologia da rede e não realiza nenhum tipo de verificação quanto à capacidade de *buffer* dos nós e largura de banda dos enlaces.

Outro algoritmo, denominado MaxProp, foi proposto em [Burgess et al. 2006] e utiliza informações de histórico de contatos para determinar a prioridade das mensagens a serem transmitidas. Um esquema de propagação de mensagens de controle para confirmação de recebimento também é implementado juntamente com uma política de replicação de mensagens e de exclusão de réplicas. As restrições de capacidade de *buffer* dos nós e largura de banda dos enlaces são consideradas, mas nenhuma informação sobre o estado da rede é conhecida antecipadamente.

O algoritmo Rapid foi desenvolvido e avaliado em [Balasubramanian et al. 2007] e tem o objetivo de rotear as mensagens até o destino por meio da replicação destas nos nós intermediários. Para evitar a sobrecarga de mensagens na rede implementou-se um mecanismo que determina se uma mensagem deve ser replicada ou removida em determinados nós intermediários. Este algoritmo não necessita de nenhuma informação prévia sobre o estado da rede, porém, utiliza informações relativas ao histórico desta para estimar novas alterações na topologia. Uma avaliação foi realizada utilizando os *traces* da rede veicular *DieselNet*, onde o Rapid foi comparado com os algoritmos MaxProp [Burgess et al. 2006], Spray and Wait [Spyropoulos et al. 2005] e PROPHET [Lindgren et al. 2004] utilizando distintas cargas de mensagens. Outro algoritmo, denominado Random, também foi implementado pelo autor com o intuito de avaliar o desempenho da entrega de mensagens quando aplicado um mecanismo de duplicação de mensagens de forma randômica entre os vizinhos. Foi verificado também o desempenho do Rapid comparado a uma solução ótima obtida através da formulação em programação linear elaborada também neste trabalho.

O algoritmo NECTAR proposto em [Oliveira e Albuquerque 2009] utiliza o conceito de índice de vizinhança, considerando que os nós movimentam-se de forma que existe certa probabilidade que vizinhos possam ser reencontrados. Políticas de escalo-

namento e descarte de mensagens são desenvolvidas e um mecanismo de replicação de mensagens é implementado. O algoritmo não considera para encaminhamento das mensagens nenhuma informação sobre o estado futuro da rede. Uma avaliação é realizada comparando os resultados do NECTAR com os obtidos pelos algoritmos Epidemic Routing [Vahdat e Becker 2000] e PROPHET [Lindgren et al. 2004] considerando tamanhos distintos de *buffer* e verificando a quantidade de mensagens entregues e o número de saltos realizados pelas mesmas.

Três algoritmos distribuídos para construção de tabelas de roteamento em DTNs previsíveis foram propostos em [Santos et al. 2008]. Em um dos algoritmos, o *Distributed Shortest Journey* (DSJ), cada nó calcula a tabela de roteamento dele para todos os outros nós considerando o menor número de saltos. No outro, denominado *Distributed Earliest Journey* (DEJ), as tabelas são construídas objetivando a chegada mais cedo da informação ao nó de destino. Por último, o algoritmo *Distributed Fastest Journey* (DFJ), realiza a construção das tabelas levando-se em consideração as jornadas mais rápidas para chegar até os destinos. Em todos os algoritmos a construção da tabela em cada nó é realizada à medida que os enlaces para os nós vizinhos tornam-se disponíveis e novas informações sobre o estado da rede são obtidas. As tabelas geradas mantêm, para cada nó de destino, uma lista ordenada dos intervalos de tempo de disponibilidade dos enlaces adjacentes. Cada intervalo é único na lista e determina qual vizinho deve receber a mensagem para posterior encaminhamento para cada nó de destino. Os algoritmos realizam um filtro nos instantes de tempo de disponibilidade dos enlaces adjacentes para evitar o envio desnecessário de mensagens de controle pelos canais de comunicação.

Em [Argolo et al. 2009], além do algoritmo AJRP, também é proposto um modelo de Programação Linear Inteira para DTNs baseado na abordagem *multi-commodities flow*. Esta formulação matemática difere das demais [Jain et al. 2004, Balasubramanian et al. 2007], pois a complexidade para encontrar a solução ótima é diretamente proporcional ao número de nós e não ao número de mensagens da rede. Uma comparação dos resultados do AJRP com a solução ótima foi realizada considerando as limitações de largura de banda dos enlaces e de capacidade de armazenamento dos nós. A avaliação mostra que o AJRP obteve um desempenho satisfatório, entregando cerca de 96% da demanda de mensagens quando atribuída baixa carga de mensagens na rede e cerca de 83% quando submetida a uma alta carga.

3. Modelo

DTNs previsíveis podem considerar várias informações conhecidas antecipadamente, tais como: a disponibilidade de contato entre os nós da rede ao longo do tempo; as demandas de mensagens de cada nó; e a capacidade de armazenamento destes. No entanto, a obtenção a *priori* de todo este conhecimento é praticamente inviável devido ao tamanho da rede ou até mesmo a própria dinâmica de geração das mensagens.

O problema de roteamento em DTNs consiste em entregar as mensagens aos destinos de acordo com uma métrica pré-determinada, levando-se em conta as restrições de disponibilidade e capacidade dos enlaces para transmissão das mensagens, assim como as limitações de armazenamento destas pelos nós da rede. As desconexões presentes nestas redes podem implicar na inexistência de uma rota fim-a-fim entre a origem e o destino.

O presente trabalho utiliza o modelo para DTNs proposto em [Argolo et al. 2009],

que considera que apenas os períodos de disponibilidade dos enlaces adjacentes são conhecidos previamente por cada nó. Para contornar as limitações de conectividade o modelo utiliza o conceito de jornada, que são definidas como rotas construídas considerando os instantes de tempo de existência dos enlaces. Desta forma, os nós intermediários armazenam em seus *buffers* as mensagens recebidas e as encaminham em momentos adequados, evitando que estas sejam roteadas para enlaces que estavam disponíveis apenas no passado. A Figura 1 ilustra uma rede onde as jornadas válidas para envio de mensagens do nó A para o nó C, devem considerar o nó B como intermediário. Ou seja, o nó A deverá enviar as mensagens para B entre os instantes 1 e 10, o nó B armazenará as mensagens em seu *buffer* e poderá encaminhá-las para o nó C a partir do instante 20.

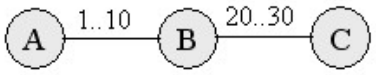


Figura 1. Exemplo de jornada

O roteamento das mensagens na rede segue uma métrica específica cujo objetivo é mensurar a qualidade da solução obtida. Exemplos de métricas que podem ser adotadas são: tempo médio ou máximo para entrega das mensagens; percentual de mensagens entregues dentro de um prazo pré-estabelecido; e número máximo de mensagens entregues [Ramanathan et al. 2007, Balasubramanian et al. 2007]. O presente trabalho optou por explorar este último critério, pois este representa um dos objetivos mais básicos e importantes com relação à entrega de mensagens em uma DTN.

Uma arquitetura para DTN com contatos programados ou previsíveis deve considerar sincronismo de tempo entre os nós da rede, conforme descrito na *Request For Comments* [Cerf 2007] publicada no *Internet Engineering Task Force* (IETF). No modelo em questão, foi definido que cada período de tempo é denominado pulso. Portanto, a cada pulso p os nós podem criar, receber, processar e enviar mensagens. Considera-se que a comunicação entre nós adjacentes é realizada dentro de um pulso, ou seja, o envio e recebimento de mensagens de controle e de aplicação não podem ultrapassar o término do pulso em que estas tiveram sua transmissão iniciada. Assume-se ainda que a fragmentação das mensagens não é admitida e que os canais de comunicação obedecem a ordem FIFO (*first-in-first-out*). Ressalta-se que os nós da rede usam esta informação para controle síncrono do tempo e verificação de quando cada enlace adjacente está ativo para comunicação.

DTNs podem apresentar propriedades cíclicas, isto é, após o último pulso a contagem é reiniciada e a execução retorna para o primeiro pulso, onde volta-se novamente a topologia de rede inicial. Para efeito de avaliação admitiu-se apenas a execução do primeiro ciclo e, por conseguinte, as mensagens não entregues neste período são descartadas.

O modelo estabelece que uma DTN previsível pode ser representada por um grafo orientado e subdividido em pulsos, isto é, períodos de tempo de mesma dimensão. Cada nó da rede possui identificação distinta e é representado por vértices em diferentes pulsos. Cada enlace é simbolizado por dois arcos de capacidade finita, onde um deles conecta um vértice v_1 no pulso t_x ao v_2 em t_{x+1}, enquanto o outro arco conecta o vértice v_2 no pulso t_x

ao v_1 em t_{x+1}. No caso do *buffer*, os vértices de origem e destino referenciam o mesmo nó da rede. Considera-se ainda que a largura de banda disponível é independente para cada um dos contatos, isto é, não é admitido o compartilhamento dos canais de comunicação estabelecidos simultaneamente com nós vizinhos.

A Figura 2 mostra um exemplo do modelo proposto. O grafo ilustrado contém 4 nós (a, b, c e d) indexados pelos instantes de tempo e seu ciclo de execução ocorre a cada 4 pulsos. Observa-se que entre os pulsos 1 e 2 os enlaces entre os nós a e b e entre os nós c e d estão disponíveis. No entanto, entre os instantes 2 e 3 apenas a comunicação entre a e b permanece disponível. Como pode ser notado, os vértices a e d não possuem conectividade, mas o d é alcançável por a através do vértice intermediário b. Para tanto, basta que, no pulso 1, a encaminhe a mensagem para b, que por sua vez deverá armazenar a mensagem no pulso 2 e reencaminhá-la para d no pulso 3. Verifica-se que a recíproca não é verdadeira, pois não existe jornada partindo de d até a.

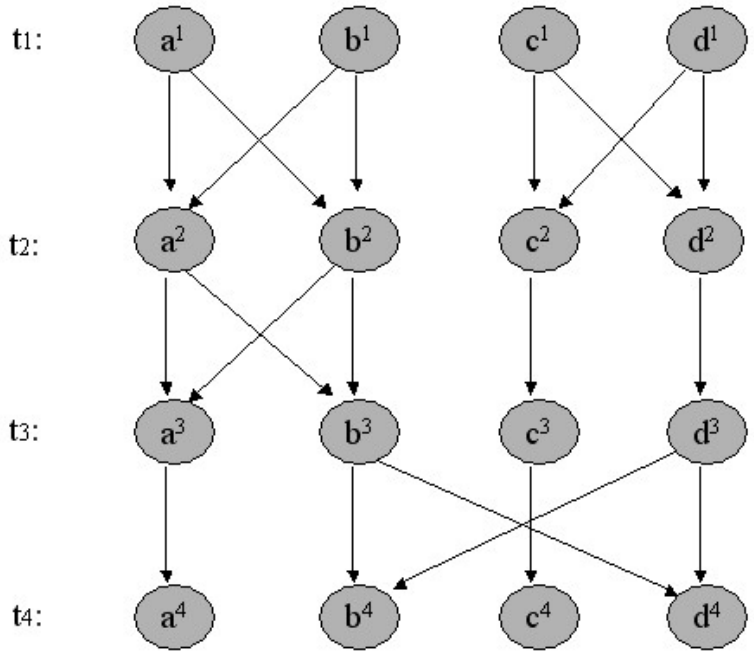


Figura 2. Exemplo de representação de uma DTN

4. O algoritmo MP-AJRP

Esta seção apresenta o algoritmo distribuído *Multi-Priority Alternative Journey Routing Protocol* (MP-AJRP) que tem como objetivo realizar o roteamento de mensagens em DTNs considerando as restrições de largura de banda dos enlaces e de capacidade de armazenamento dos nós. A estratégia de roteamento distribuído proposta é composta de duas etapas. Primeiramente, executa-se um algoritmo para construção de tabelas de roteamento em cada nó da rede, mapeando as jornadas viáveis até os destinos. A partir destas tabelas, realiza-se o envio e recebimento das mensagens empregando um mecanismo de escolha de jornadas alternativas evitando, assim, a sobrecarga de mensagens em determinados nós da rede.

Conforme mencionado na Seção 3, um ciclo é dividido em instantes de tempo denominados pulsos. Em cada pulso, quando necessário, os nós enviam mensagens de

controle e aplicação para os vizinhos que estão com enlace ativo e processam as mensagens recebidas. Assume-se que a transmissão é realizada dentro de um pulso e que os tempos de processamento das mensagens são desprezíveis. Classificam-se as mensagens como referentes à aplicação, ou seja, advindas de uma camada de *software* de nível superior ao roteamento, ou como mensagens de controle, que servem como auxílio para os algoritmos executarem o roteamento. Define-se ainda que mensagens são perdidas apenas quando o nó receptor atingiu sua capacidade máxima de armazenamento ou quando as mesmas não alcançaram o destino até o último pulso t_f, ou seja, permaneceram no *buffer* de algum nó intermediário.

Uma vez construídas as tabelas de roteamento, cada nó tem conhecimento dos momentos adequados para envio de mensagens de aplicação para os demais nós da rede. No entanto, à medida que mais mensagens são geradas, tanto a utilização dos canais de comunicação como a dos *buffers* dos nós intermediários aumentam, congestionando as jornadas e provocando o atraso na entrega ou até a perda de mensagens.

Em virtude disto, o MP-AJRP faz uso de mensagens de controle com o intuito de balancear a carga de mensagens entre as jornadas disponíveis para cada nó da rede. Isto é feito através do envio do percentual de ocupação do *buffer* dos nós para seus vizinhos. A partir desta informação, os nós escolhem como preferencial uma jornada que passa pelo vizinho com maior disponibilidade de armazenamento de mensagens no momento. Isto posto, as mensagens de aplicação são encaminhadas para jornadas que apresentam menos congestionamento, prevenindo eventuais gargalos na rede. Além disso, as tabelas de roteamento utilizadas consideram os menores números de saltos que a mensagem de aplicação realiza até chegar ao destino, reduzindo a utilização de recursos da rede e, consequentemente, a perda de mensagens devido às limitações de armazenamento dos nós e à capacidade de trasmissão dos enlaces da rede.

Para avaliação do MP-AJRP foi empregada uma estrutura de dados para mapear os instantes de tempo de geração das mensagens pelos nós. Embora o algoritmo não faça uso desta informação para tomada de decisões de roteamento, ou seja, os nós não detêm conhecimento *a priori* da lista de demandas futuras, esta informação é necessária para simulação da geração das mensagens nos momentos pré-estabelecidos nos experimentos.

Os dados utilizados pelo Algoritmo 1, assim como as principais variáveis e suas inicializações, estão descritos a seguir.

O algoritmo MP-AJRP é síncrono e sua apresentação é feita através de dois eventos que, por sua vez, demandam uma ação específica. O primeiro, refere-se à contagem do tempo, onde, a cada pulso p, mensagens de aplicação podem ser recebidas pelos nós e armazenadas na variável $buffer_i$ (linhas 5 a 12). Mensagens também podem ser geradas neste momento de acordo com a variável $demandList_i$ (linhas 13 a 17). O envio destas, e de outras que estiverem no *buffer* do nó, é feito caso o enlace com o vizinho apropriado esteja disponível e sua capacidade ainda não tenha sido esgotada (linhas 22 a 26). As funções $enqueue(message, pol)$ (linha 8 e 15) e $dequeue(j, pol)$ (linha 22) são empregadas, respectivamente, para incluir uma mensagem no *buffer* do nó e para extrair uma mensagem que tenha um destino tal que a jornada para este passe pelo vizinho j, ambas utilizando uma determinada política pol. Esta política pode ser a FIFO, onde a escolha da mensagem no *buffer* é realizada seguindo a ordem em que foram inseridas, a HOP,

Dados

N	=	conjunto de nós da rede
M	=	conjunto de enlaces da rede
R	=	conjunto de jornadas para cada destino
Y	=	conjunto de intervalos de tempo
pol	=	política de seleção das mensagens do *buffer* - FIFO, HOP ou MEET
$Neig_i$	=	conjunto de vizinhos do nó i
$myRank_i$	=	identificação do nó i

Variáveis

p	=	0
SET_i	=	$nulo$ (conjunto de mensagens enviadas pelo nó i)
$message$	=	$nulo$
$setSize$	=	0
$buffer_i$	=	$nulo$
$bufferUsage_i$	=	0
$bufferSize_i$	=	tamanho do *buffer* do nó i
$status_i[j]$	=	0, $\forall\, j \in Neig_i$
$vecInter_i[j]$	=	(vizID, capacidade, up[], down[]), $\forall\, n_j \in Neig_i$ (vetor com informações sobre o enlace entre os nós i e j)
$vecUsage_i[j]$	=	$nulo$, $\forall\, j \in Neig_i$
$routeTable_i[j][r][y]$	=	$nulo$, $\forall\, j \in N$, $r \in R$ e $y \in Y$
$demandList_i[j][y]$	=	conjunto de mensagens para envio, tal que $j \in N$ e $y \in Y$

onde as mensagens são ordenadas priorizando as que são destinadas a nós que demandem o menor número de saltos, e a MEET, onde a ordenação das mensagens é feita beneficiando aquelas destinadas a nós que possuem o menor número de intervalos de tempo de disponibilidade durante sua jornada. Estas informações são obtidas através da tabela de roteamento gerada pelo algoritmo descrito em [Santos et al. 2008].

O segundo evento trata das mensagens de controle recebidas. A ação tomada neste caso é verificar para cada destino da tabela de roteamento, representada pela variável $routeTable_i$, qual jornada está menos congestionada e considerar esta como preferencial para o roteamento das próximas mensagens (linhas 36 a 43).

4.1. Complexidade do algoritmo

O algoritmo MP-AJRP termina sua execução após processamento do último pulso $t_f \in T$, isto é, a complexidade de tempo é da ordem de $O(T)$. A quantidade total de pulsos (T) depende da DTN em questão. Ou seja, de acordo com os tempos de disponibilidade dos enlaces da rede será identificada a quantidade de pulsos de execução.

Com relação a quantidade de mensagens de controle enviadas pelo MP-AJRP, considerando que cada enlace $e \in M$ possua y_e ativações, e que cada período iniciado por um y possua p_{ey} pulsos, o número de mensagens de controle enviado é $\sum_{e=1}^{|M|} \sum_{y=1}^{|y_e|} p_{ey}$.

A tabela de roteamento mapeia os Y intervalos de cada conjunto R de jornadas possíveis para cada um dos $N-1$ nós de destino. Assim sendo, a complexidade de

```
1  Algorithm MP − AJRP(verInter, routeTable, demList)
2  INPUT:
3    p > 0, SET_j ∈ MSG_i(p), origem_i(SET_j) = (n_i, n_j);
4  ACTION:
5  forall message ∈ SET_j do
6      if (message.destId ≠ myRank_i) then
7          if (bufferUsage_i < bufferSize_i) then
8              enqueue(message, pol);
9              bufferUsage_i ← bufferUsage_i + 1;
10         end
11     end
12 end
13 forall message ∈ demandList_i[j][y], j ∈ N, p = y, y ∈ Y do
14     if (bufferUsage_i < bufferSize_i) then
15         enqueue(message, pol);
16         bufferUsage_i ← bufferUsage_i + 1;
17     end
18 end
19 forall active (i, j) in p, j ∈ Neig_i do
20     SET_i[k] ← nulo, ∀ k < setSize;
21     setSize ← 0;
22     while ((message = dequeue(j, pol)) ≠ nulo) AND
       (setSize ≤ vecInter_i(j).capacidade) do
23         bufferUsage_i ← bufferUsage_i − 1;
24         setSize ← setSize + 1;
25         SET_i[setSize] ← message;
26     end
27     SEND SET_i to n_j;
28     percent ← (bufferUsage_i/bufferSize_i) ∗ 100;
29     SEND Control(percent) to n_j;
30 end
31 INPUT:
32   p > 0, Control_i(percent), origem_i = n_j;
33 ACTION:
34 vecUsage_i[j] ← percent;
35 choice ← vecUsage_i[0];
36 forall k ∈ N do
37     forall r ∈ R, y ∈ Y, y > p do
38         if vecUsage_i[j] < choice then
39             choice ← vecUsage_i[j];
40         end
41     end
42     routeTable_i[k].choice ← choice;
43 end
```

Algorithm 1: Algoritmo MP-AJRP

armazenamento desta estrutura de dados é de $O(NRY)$.

5. Resultados experimentais

A avaliação dos algoritmos foi realizada através de simulações utilizando um ambiente com 5 computadores conectados em LAN. Cada máquina possui processador Intel Pentium IV 2.8GHz e memória RAM de 512MB, rodando o sistema operacional Ubuntu V3. A implementação foi feita em linguagem C utilizando a biblioteca MPI para troca de mensagens. Cada nó das instâncias de rede utilizadas nos testes foi simulado por um processo MPI e alocado em uma das máquinas da LAN.

Nesta seção são apresentados os resultados com relação ao encaminhamento das mensagens até os destinos, considerando as restrições de largura de banda dos enlaces e da capacidade de armazenamento dos nós.

Experimentos foram realizados com o MP-AJRP utilizando os *traces* gerados pela rede veicular *DieselNet* entre 14 de fevereiro e 15 de maio de 2007, onde foram consideradas as mesmas capacidades de *buffer* dos nós, larguras de banda dos enlaces e demandas de mensagens utilizadas em [Balasubramanian et al. 2007]. A Tabela 1 apresenta estas informações. Para evitar a geração de mensagens para destinos que nunca seriam alcançados foram considerados como potenciais destinos apenas os ônibus circulantes no dia corrente. Definiu-se como carga de mensagem a quantidade de mensagens geradas a cada hora para todos os destinos disponíveis no *trace* do dia corrente.

Primeiramente, foi avaliado apenas o algoritmo MP-AJRP utilizando três políticas distintas para seleção de mensagens no *buffer*, a saber: FIFO, HOP e MEET. Na primeira, a escolha da mensagem no *buffer* é realizada seguindo a ordem em que foram inseridas. Na segunda, as mensagens são ordenadas priorizando as que são destinadas a nós que demandem o menor número de saltos. Na última, a ordenação das mensagens é feita beneficiando aquelas destinadas a nós que possuem o menor número de intervalos de tempo de disponibilidade durante sua jornada.

Nas execuções do MP-AJRP foram utilizadas as tabelas de roteamento geradas pelo algoritmo DSJ, mantendo até três jornadas por intervalo $(R = 3)$ para cada destino.

Capacidade de armazenamento de cada ônibus	40GB
Quantidade total de ônibus	40
Quantidade média de ônibus por dia	20
Hora de início da avaliação para cada dia de execução	8 horas
Hora de término da avaliação para cada dia de execução	19 horas
Hora de início da geração de mensagens	8 horas
Hora de término da geração de mensagens	15 horas
Tempo entre a geração das cargas de mensagens	1 hora
Tempo de cada pulso	1 segundo

Tabela 1. Descrição do ambiente de teste com os *traces* da DieselNet

A Figura 3 ilustra a porcentagem de entrega de mensagens para cada um dos destinos. Como pode ser observado, a seleção através do menor número de saltos conseguiu entregar, para todas as cargas de mensagem avaliadas, o maior número de mensagens aos destinos, seguido pela política MEET e por último a FIFO. Observa-se também que estas duas últimas políticas tiveram desempenhos muito próximos. A explicação para o melhor resultado obtido pela política HOP é exatamente a priorização das mensagens reduzindo

a utilização dos recursos da rede. Ou seja, primeiro envia-se as mensagens cujos destinos são os próprios vizinhos. Em seguida, as mensagens cujos destinos são os vizinhos dos vizinhos são selecionadas, e assim sucessivamente.

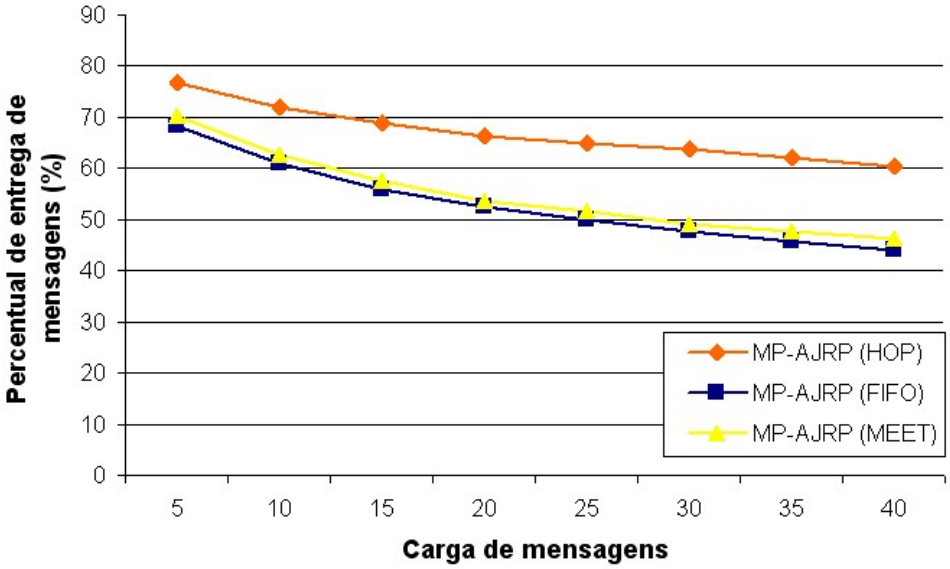


Figura 3. Entrega de mensagens variando o algoritmo de seleção de roteamento

Os resultados obtidos pelo MP-AJRP utilizando a política HOP foram comparados com os obtidos pelos algoritmos RAPID, MaxProp, Spray and Wait, Random e PROPHET, que são algoritmos que foram propostos nos últimos anos com o intuito de otimizar o roteamento de mensagens em DTNs e foram recentemente implementados e avaliados em [Balasubramanian et al. 2007].

A Figura 4 demonstra que o MP-AJRP, mesmo sem realizar nenhum tipo de replicação de mensagens, ou seja, apenas encaminhando cada mensagem a um nó intermediário até alcançar o destino, obteve desempenho superior ao algoritmo Random para todas as cargas de mensagem consideradas. Além disso, observa-se que a diferença na porcentagem de entrega de mensagens pelo MP-AJRP e pelo algoritmo Spray and Wait é relativamente pequena, sendo o MP-AJRP melhor com as cargas de 5 e 40 mensagens, e pior nas demais. Nota-se, contudo, que a curva apresentada pelo algoritmo Spray and Wait apresenta tendência de queda mais acentuada a partir da carga de 35 mensagens, enquanto que a tendência do MP-AJRP é de decréscimo mais suave. Ressalta-se que o algoritmo PROPHET não aparece neste gráfico devido ao seu desempenho muito inferior comparado ao algoritmo Random, conforme já descrito em [Balasubramanian et al. 2007].

Como pode ser observado, o MP-AJRP não conseguiu superar os algoritmos Rapid e MaxProp para as instâncias utilizadas. Um dos motivos para isso é que estes dois algoritmos fazem duplicação de mensagens e usam mecanismos para gerenciamento das réplicas. Por demandar mais recursos da rede, estas estratégias obtiveram melhor desempenho para a topologia de rede avaliada, favorecendo o percentual de entrega das mensagens. No entanto, os resultados apontam que com algum tipo de replicação, menos onerosa do que as empregadas em algoritmos da literatura, o MP-AJRP poderia obter melhores resultados frente aos algoritmos avaliados neste trabalho.

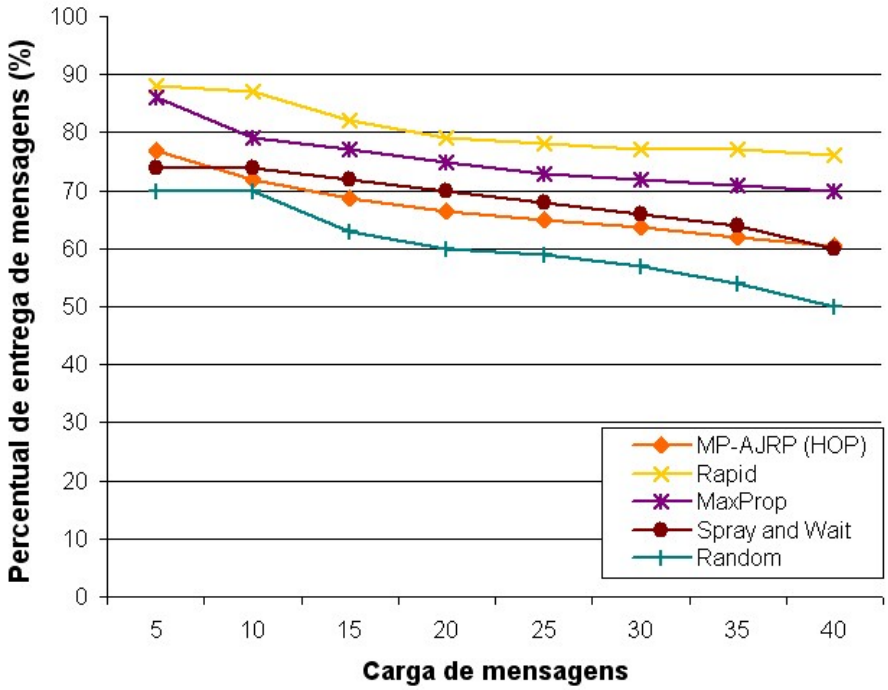


Figura 4. Entrega de mensagens de variados algoritmos

6. Conclusões e trabalhos futuros

Neste trabalho foi proposto o algoritmo distribuído *Multi-Priority Alternative Journey Routing Protocol* (MP-AJRP) que foi desenvolvido com o objetivo de maximizar a entrega de mensagens aos destinos considerando as restrições de largura de banda dos enlaces e as limitações de capacidade de armazenamento dos nós, utilizando um mecanismo de seleção de jornadas alternativas menos congestionadas. Além disso, diferentes políticas de priorização de envio das mensagens do *buffer* foram implementadas e avaliadas. Este algoritmo foi avaliado utilizando *traces* gerados pela rede veicular *DieselNet* e verificou-se que, embora não realize nenhum tipo de replicação de mensagens, seus resultados superaram os de alguns algoritmos de roteamento em DTNs que consideram tal funcionalidade. Vale ressaltar ainda que políticas menos onerosas de replicação de mensagens podem ser incluídas ao MP-AJRP a fim de otimizar os resultados obtidos pelo algoritmo. Tais modificações podem reduzir a diferença de desempenho ou até superar os resultados dos algoritmos que foram aqui comparados com o MP-AJRP.

Como continuação deste trabalho, pretende-se desenvolver algoritmos de encaminhamento que adotam outras métricas de roteamento como, por exemplo, a de entrega das mensagens considerando seus *deadlines*. Além disso, pretende-se desenvolver mecanismos para melhorar o desempenho do algoritmo de encaminhamento usando, para isso, replicação de mensagens de aplicação e incorporação de mais informações relevantes nas mensagens de controle. Outros pontos importantes a serem pesquisados são a avaliação do desempenho do algoritmo proposto neste trabalho em outras topologias e também a comparação deste com outros algoritmos existentes na literatura.

Referências

Argolo, A., Dolejsi, L. D. A., E.Uchoa, e Subramanian, A. (2009). Roteamento em redes tolerantes a atrasos e desconexões com restrições de buffer e largura de banda. In *XLI Simpósio Brasileiro de Pesquisa Operacional*.

Balasubramanian, A., Levine, B., e Venkataramani, A. (2007). Dtn routing as a resource allocation problem. In *ACM SIGCOMM*, pages 373–384.

Bui-Xuan, B., Ferreira, A., e Jarry, A. (2003). Evolving graphs and least cost journeys in dynamic networks. In *Modeling and Optimization in Mobile, Ad-Hoc, and Wireless Networks*, pages 141–150.

Burgess, J., Gallagher, B., Jensen, D., e Levine, B. N. (2006). Maxprop: Routing for vehicle-based disruption-tolerant networks. In *IEEE Infocom*, pages 1–11.

Cerf, V. (2007). Rfc 4838: Dtn architecture. Technical report, IETF.

Chen, C. (2005). Advanced routing protocol for satellite and space networks. Master's thesis, Georgia Institute of Technology.

Jain, S., Fall, K., e Patra, S. (2004). Routing in a delay tolerant network. In *ACM SIGCOMM*, pages 145–158.

Lindgren, A., Doria, A., e Schelén, O. (2004). Probabilistic routing in intermittently connected networks. In *SAPIR Workshop*, pages 239–254.

Merugu, S., Ammar, M., e Zegura, E. (2004). Routing in space and time in networks with predictable mobility. Technical report, Georgia Institute of Technology.

Oliveira, C. T. e Duarte, O. C. M. B. (2007). Uma análise da probabilidade de entrega de mensagens em redes tolerantes a atrasos e desconexões. In *XXV Simpósio Brasileiro de Redes de Computadores*, pages 293–305.

Oliveira, E. C. R. e Albuquerque, C. V. N. (2009). Nectar: A dtn routing protocol based on neighborhood contact history. In *24th ACM Symposium on Applied Computing*, pages 359–365.

Peleg, D. (2000). *Distributed Computing: a locality-sensitive approach (Monographs on Discrete Mathematics and Applications)*. Society for Industrial Mathematics.

Ramanathan, R., Basu, P., e Krishnan, R. (2007). Towards a formalism for routing in challenged networks. In *ACM MobiCom workshop on Challenged Networks*, pages 3–10.

Santos, A., Rocha, G., Drummond, L., e Gorza, M. (2008). Algoritmos distribuídos para roteamento em redes tolerantes a atrasos e desconexões. In *Workshop em Sistemas Computacionais de Alto Desempenho*, pages 35–42.

Spyropoulos, T., Psounis, K., e Raghavendra, C. S. (2005). Spray and wait: An efficient routing scheme for intermittently connected mobile networks. In *ACM WDTN*, pages 252–259.

Vahdat, A. e Becker, D. (2000). Epidemic routing for partially connected ad hoc networks. Technical report, Duke University.

Índice por Autor