SBRC 2016

XXXIV Simpósio Brasileiro
de Redes de Computadores
e Sistemas Distribuídos

30 de maio a 03 de junho de 2016
Salvador - Bahia - Brasil
www.sbrc2016.ufba.br

Foto Pierre Verger © Fundação Pierre Verger

ANAIS **WTF 2016**

ORGANIZAÇÃO

computação UFBA

UFBA 70 ANOS

INSTITUTO FEDERAL BAHIA

XXXIV Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos

SBRC 2016

30 de maio a 03 de junho de 2016
Salvador - Bahia - Brasil
www.sbrc2016.ufba.br

Anais do WTF 2016

Workshop de Testes e Tolerância a Falhas

Editora

Sociedade Brasileira de Computação (SBC)

Organização

Luiz Antonio Rodrigues (UNIOESTE)
Michele Nogueira Lima (UFPR)
Fabíola Gonçalves Pereira Greve (UFBA)
Allan Edgard Silva Freitas (IFBA)

Realização

Universidade Federal da Bahia (UFBA)
Instituto Federal da Bahia (UFBA)

Promoção

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

Capa: Gilson Rabelo (UFBA)
Produção Editorial: Antonio Augusto Teixeira Ribeiro Coutinho (UEFS)

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

Workshop de Testes e Tolerância a Falhas (6: 2016: Salvador, BA).

Anais / XVII Workshop de Testes e Tolerância a Falhas; organizado por Luiz Antonio Rodrigues, Michele Nogueira Lima, Fabíola Gonçalves Pereira Greve, Allan Edgard Silva Freitas - Porto Alegre: SBC, 2016

112 p. il. 21 cm.
Vários autores
Inclui bibliografias
1. Redes de Computadores. 2. Sistemas Distribuídos. I. Rodrigues, Luiz Antonio II. Lima, Michele Nogueira III. Greve, Fabíola Gonçalves Pereira IV. Freitas, Allan Edgard Silva V. Título.

Sociedade Brasileira da Computação

Contato
Av. Bento Gonçalves, 9500
Setor 4 - Prédio 43.412 - Sala 219
Bairro Agronomia
91.509-900 – Porto Alegre RS
CNPJ: 29.532.264/0001-78
http://www.sbrc.org.br

Laboratório Nacional de Redes de Computadores (LARC)

Diretora do Conselho Técnico-Científico
Rossana Maria de C. Andrade (UFC)

Vice-Diretor do Conselho Técnico-Científico
Ronaldo Alves Ferreira (UFMS)

Diretor Executivo
Paulo André da Silva Gonçalves (UFPE)

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

Membros Institucionais
SESU/MEC, INPE/MCT, UFRGS, UFMG, UFPE, UFCG (ex-UFPB Campus Campina Grande), UFRJ, USP, PUC-Rio, UNICAMP, LNCC, IME, UFSC, UTFPR, UFC, UFF, UFSCar, IFCE (CEFET-CE), UFRN, UFES, UFBA, UNIFACS, UECE, UFPR, UFPA, UFAM, UFABC, PUCPR, UFMS, UnB, PUC-RS, UNIRIO, UFS e UFU.

Contato
Universidade Federal de Pernambuco - UFPE
Centro de Informática - CIn
Av. Jornalista Anibal Fernandes, s/n
Cidade Universitária
50.740-560 - Recife - PE
http://www.larc.org.br

Organização do SBRC 2016

Coordenadores Gerais
Fabíola Gonçalves Pereira Greve (UFBA)
Allan Edgard Silva Freitas (IFBA)
Romildo Martins Bezerra (IFBA) - In Memoriam

Coordenadores do Comitê de Programa
Antonio Marinho Pilla Barcellos (UFRGS)
Antonio Alfredo Ferreira Loureiro (UFMG)

Coordenador de Palestras e Tutoriais
Francisco Vilar Brasileiro (UFCG)

Coordenador de Painéis e Debates
Carlos André Guimarães Ferraz (UFPE)

Coordenadores de Minicursos
Lau Cheuck Lung (UFSC)
Frank Siqueira (UFSC)

Coordenadora de Workshops
Michele Nogueira Lima (UFPR)

Coordenador do Salão de Ferramentas
Daniel Macêdo Batista (USP)

Comitê de Organização Local
Adolfo Duran (UFBA)
Allan Freitas (IFBA)
Antonio Augusto Teixeira Ribeiro Coutinho (UEFS)
Cátia Khouri (UESB)
Fabíola Greve (UFBA)
Flávia Maristela Nascimento (IFBA)
Gustavo Bittencourt (UFBA)
Ítalo Valcy (UFBA)
Jauberth Abijaude (UESC)
Manoel Marques Neto (IFBA)
Marco Antônio Ramos (UESB)
Marcos Camada (IF BAIANO)
Marcos Ennes Barreto (UFBA)
Maycon Leone (UFBA)
Rafael Reale (IFBA)
Renato Novais (IFBA)
Ricardo Rios (UFBA)
Romildo Bezerra (IFBA) - In Memoriam
Sandro Andrade (IFBA)
Vinícius Petrucci (UFBA)

Comitê Consultivo
Jussara Almeida (UFMG), Coordenadora
Elias Procópio Duarte Jr. (UFPR)
José Rezende (UFRJ)
Jacir Luiz Bordim (UnB)
Rafael Timóteo de Sousa Júnior (UnB)
William Ferreira Giozza (UnB)
Carlos André Guimarães Ferraz (UFPE)
José Augusto Suruagy Monteiro (UFPE)

Mensagem da Coordenadora de Workshops

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

Em resposta à chamada, recebemos nove propostas de alta qualidade, das quais sete estão sendo de fato organizadas nesta edição do SBRC em Salvador. Dentre as propostas aceitas, quatro reforçaram workshops tradicionais do SBRC, considerados parte do circuito nacional de divulgação científica nas várias subáreas de Redes de Computadores e Sistemas Distribuídos, como o WGRS (Workshop de Gerência e Operação de Redes e Serviços), o WTF (Workshop de Testes e Tolerância a Falhas), o WCGA (Workshop em Clouds, Grids e Aplicações) e o WP2P+ (Workshop de Redes P2P, Dinâmicas, Sociais e Orientadas a Conteúdo). Além disso, três propostas apoiam a consolidação de subáreas mais recentes, tais como as propostas de organização do WPEIF (Workshop de Pesquisa Experimental da Internet do Futuro), do WoSiDA (Workshop de Sistemas Distribuídos Autonômicos) e do WoCCES (Workshop de Comunicação de Sistemas Embarcados Críticos).

Esperamos que 2016 seja mais um ano de sucesso para os workshops do SBRC, contribuindo como importantes fatores de agregação para os avanços promovidos pela comunidade científica da área de Redes de Computadores e Sistemas Distribuídos no Brasil.

Aproveitamos para agradecer o inestimável apoio recebido de diversos membros da comunidade e, em particular a cada coordenador de workshop, pelo brilhante trabalho, e a Organização Geral do SBRC 2016 pelo profissionalismo e comprometimento.

A todos, um excelente SBRC em Salvador!

Michele Nogueira
Coordenadora de Workshops do SBRC 2016

Mensagem do Coordenador do WTF 2016

A área de pesquisa em testes e tolerância a falhas visa desenvolver técnicas para a construção de sistemas computacionais confiáveis e de alta disponibilidade. Neste sentido, o Workshop de Testes e Tolerância a Falhas (WTF) é um fórum criado em 1998 para discussões e troca de ideias sobre trabalhos teóricos e práticos relacionados a testes e a tolerância a falhas desenvolvidos tanto na academia como na indústria. A programação desta 17ª edição inclui 9 artigos (7 completos e 2 resumos estendidos) selecionados pelo Comitê de Programa e uma palestra convidada. O artigos foram agrupados em três sessões técnicas: Medidas de Confiabilidade e Segurança, Ferramentas e Sistemas para Testes e Tolerância a falhas e Algoritmos distribuídos. A palestra convidada será ministrada pelo professor Sérgio Gorender, da Universidade Federal da Bahia (UFBA), abordando modelos híbridos para sistemas distribuídos e suas aplicações. Agradeço a todos os membros do Comitê de Programa pelo comprometimento e cumprimento dos prazos de retorno das avaliações, todas muito criteriosas e construtivas. Obrigado especialmente pela ajuda na elaboração do projeto, na divulgação dos prazos, na condução das revisões, na indicação de novos membros e no processo de revisão às cegas. Obrigado também aos membros da Comissão Especial de Tolerância a Falhas (CE-TF) da Sociedade Brasileira de Computação (SBC), especialmente o prof. Sérgio Cechin, pelo apoio durante todo o processo de construção e realização desta edição. Por fim, agradeço aos organizadores do SBRC 2016, especialmente a profa. Michele Nogueira, coordenadora dos workshops, a profa. Fabíola Greve, coordenadora geral do SBRC, e a equipe do website pelo excelente trabalho, oferecendo total apoio e solução em todos os procedimentos necessários à realização deste Workshop.

Luiz Antonio Rodrigues
Chair do WTF 2016

Comitê de Programa

Alcides Calsavara (PUC-PR)
Allyson Bessani (UL - Portugal)
Ana Ambrosio (INPE)
Avelino Zorzo (PUC-RS)
Cátia Khouri (UFBA)
Daniel Batista (USP)
Daniel Cordeiro (USP)
Daniel Fernandes Macedo (UFMG)
Eduardo Alchieri (UNB)
Eduardo Bezerra (UFSC)
Eliane Martins (UNICAMP)
Elias P. Duarte Jr. (UFPR)
Fabíola Greve (UFBA)
Fernando Dotti (PUCRS)
Frank Siqueira (UFSC)
Irineu Sotoma (UFMS)
Jaime Cohen (UEPG)
João Eugenio Marynowski (UFPR)
Jose Pereira (INESC TEC e UM - Portugal)
Lau Cheuk Lung (UFSC)
Luciana Arantes (LIP6 - França)
Luiz Eduardo Buzato (UNICAMP)
Marco Vieira (UC - Portugal)
Michele Nogueira (UFPR)
Miguel Correia (UL - Portugal)
Raimundo Barreto (UFAM)
Raul Ceretta Nunes (UFSM)
Regina Moraes (UNICAMP)
Rogerio de Lemos (University of Kent - Reino Unido)
Rui Oliveira (UM - Portugal)
Sérgio Luis Cechin (UFRGS)
Sergio Gorender (UFBA)
Taisy Weber (UFRGS)
Udo Fritzke Jr. (PUC-Minas)

Sumário

WTF 2016
Sessão Técnica 1
Medidas de Confiabilidade e Segurança

Avaliação de Desempenho de Algoritmos RSA para Redes Ópticas Elásticas com Tolerância a Falhas em Cenário com Imperfeições de Camada Física

Jurandir Lacerda Jr[1,2], Alexandre Fontinele[3], Igo Moura[4], André Soares[1]

[1] Departamento de Computação
Universidade Federal do Piauí (UFPI)
Teresina – PI – Brasil

[2]Instituto Federal de Educação, Ciência e Tecnologia do Piauí (IFPI)
Corrente – PI – Brasil

[3]Centro de Informática (CIn)
Universidade Federal de Pernambuco (UFPE)
Recife – PE – Brasil

[4]Instituto Federal de Educação, Ciência e Tecnologia do Maranhão (IFMA)
Coelho Neto – MA – Brasil

jurandir.cavalcante@ifpi.edu.br

Abstract. *With the growth of Internet traffic in recent years, new technologies emerge to support the increased demand in the infrastructure of transport networks. The Elastic Optical Networks are pointed at the literature as the main technology to meet this demand. In an elastic optical network, fault tolerance becomes an important criterion to be evaluated. However, most of the works that treat this theme does not take into account the imperfections of physical layer of Elastic Optical Networks. This paper studies the impact of these imperfections of physical layer in a routing algorithm and spectrum allocation, fault-tolerant, which makes use of multipath.*

Resumo. *Com o crescimento do tráfego na Internet nos últimos anos, novas tecnologias surgem para suportar o aumento da demanda na infraestrutura de redes de transporte. As Redes Ópticas Elásticas são apontadas na literatura como a principal tecnologia para suprir esta demanda. Em uma rede óptica elástica, a tolerância a falhas torna-se um importante critério a ser avaliado. Entretanto, a maioria dos trabalhos que tratam este tema não levam em conta as imperfeições de camada física inerentes das Redes Ópticas Elásticas. Este trabalho estuda o impacto destas imperfeições de camada física em um algoritmo de roteamento e alocação de espectro, tolerante a falhas, que faz uso de técnicas multipath.*

1. Introdução

Aplicações que utilizam a infraestrutura da Internet tornam-se cada vez mais frequentes. Uma das consequências disso é o aumento da demanda por banda passante. Dessa forma, é necessário o desenvolvimento de novas tecnologias capazes de suportar esta crescente necessidade.

A tecnologia de rede óptica é apontada como a alternativa mais promissora para fazer parte da infraestrutura de rede que suporta a Internet [Chatterjee et al. 2015]. Atualmente a tecnologia DWDM (*Dense Wavelength Division Multiplexing*) é adotada para viabilizar a comunicação dentro de uma rede óptica. A tecnologia DWDM emprega a multiplexação por divisão de comprimento de onda. Com isso, o espectro óptico é dividido em vários canais de comunicação independentes, denominados comprimentos de onda. Cada comprimento de onda é utilizado para o estabelecimento de um circuito óptico [Ramaswami and Sivarajan 2009].

Entretanto, este tipo de divisão rígida pode causar um mau uso dos recursos da rede, visto que a necessidade de largura de banda de cada cliente pode variar de forma significativa. Há duas consequências que devem ser consideradas ao implantar a tecnologia DWDM em uma rede. A primeira é que caso a largura de banda exigida por um cliente seja maior do que a fixada pela tecnologia para cada canal, a demanda do cliente não é atendida. A segunda é que caso a largura de banda exigida pelo cliente seja menor que a largura disponibilizada, há um desperdício do uso destes recursos. Nesse sentido, a tecnologia OFDM (*Orthogonal Frequency-Division Multiplexing*) surge como uma solução para tal problema. A tecnologia OFDM é capaz de dividir o espectro óptico em canais de menor granularidade. Estes canais, denominados *slots*, podem ser agrupados de tal forma que se ajuste um canal de comunicação de acordo com a largura de banda requisitada. A rede que faz uso da tecnologia OFDM é conhecida como rede óptica elástica [Jinno et al. 2009].

Em uma rede óptica elástica, existem desafios relevantes que devem ser tratados, como por exemplo o problema RSA (*Routing and Spectrum Assignment*). O problema RSA consiste em realizar a escolha da melhor rota para o tráfego das informações e alocar o conjunto de *slots* necessários para atender a uma certa demanda. Entretanto, para que uma rede atinja níveis de satisfação para os clientes que a usam, outro critério deve ser levado em consideração. Tal critério remete-se à disponibilidade da rede, ou seja, o quão ela estará preparada para contornar situações de falhas. Projetar uma Rede Óptica Elástica tolerante a falhas é um desafio que a literatura aborda com frequência e relevância.

Muitos algoritmos vêm tratando tolerância a falhas nos últimos anos [Chen et al. 2015] [Shen et al. 2014] [Wang et al. 2015c]. Porém, a grande maioria destes estudos acabam avaliando seus algoritmos em cenários que negligenciam as imperfeições de camada física.

O objetivo deste trabalho é avaliar o desempenho de técnicas de tolerância a falhas, em um cenário ciente de imperfeições de camada física. Desta forma poderemos avaliar tais técnicas em um cenário mais próximo do real e mensurar qual o impacto que estas imperfeições causam no desempenho destes algoritmos. Serão usadas técnicas de simulação computacional para a análise de desempenho e a probabilidade de bloqueio por circuito será a métrica de avaliação.

O restante deste trabalho está organizado da seguinte forma. Na Seção 2 é apresentado os principais conceitos sobre redes ópticas elásticas. Uma revisão bibliográfica sobre algoritmos RSA tolerantes a falha é apresentada na Seção 3. A Seção 4 apresenta os conceitos de camada física. A avaliação de desempenho é apresentada na Seção 5. Por fim, as conclusões e trabalhos futuros são apresentadas na Seção 6.

2. Redes Ópticas Elásticas

As redes ópticas elásticas são caracterizadas pela sua capacidade em dividir os recursos espectrais em *slots* de frequência, na forma de sub-portadoras, através da multiplexação OFDM [Horota et al. 2014]. Em uma mesma fibra é possível operar com diversos circuitos ópticos. Cada circuito possui um conjunto de *slots* alocados para atender aos requisitos de banda passante. Para que um circuito óptico não interfira no sinal dos circuitos vizinhos que estão realizando transmissões simultâneas, estes são separados por um intervalo de frequência, denominado banda de guarda. Atualmente os *slots* de frequência possuem o tamanho de 12,5 GHz [Chatterjee et al. 2015].

Para viabilizar a comunicação dentro de uma rede óptica elástica é necessário determinar um caminho que os dados irão percorrer dentro da rede e especificar o conjunto de *slots* de frequências que serão utilizados. A literatura denomina RSA o problema de rotear e alocar espectro em uma rede óptica elástica [Christodoulopoulos et al. 2011]. Para satisfazer a restrição de contiguidade de espectro, um algoritmo RSA de alocação de espectro deve ser capaz de alocar um conjunto de *slots* que sejam adjacentes uns aos outros [Talebi et al. 2014]. Outra propriedade relevante que deve ser respeitada é a de continuidade do espectro óptico [Talebi et al. 2014]. Se um circuito óptico ocupa três *slots* contiguos em uma rota com três nós da rede, por exemplo, a propriedade de continuidade do espectro determina que esses mesmos *slots* deverão ser utilizados nos enlaces que conectam cada um dos nós da rota selecionada.

A medida que a rede opera, recursos são alocados e desalocados continuamente. Por conta do dinamismo no uso dos recursos há uma natural fragmentação do espectro [Talebi et al. 2014]. O problema de fragmentação do espectro influência diretamente na eficiência de uma rede óptica elástica. A Figura 1 ilustra o problema de fragmentação do espectro.

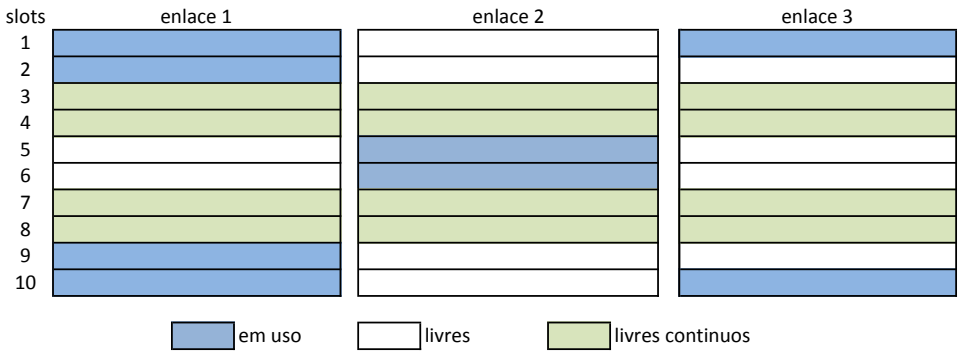


Figura 1. Cenário com fragmentação (Adaptada de [Santos 2015])

No exemplo acima ilustra-se três enlaces com *slots* de frequência livres e outros ocupados. Ao tentar estabelecer um circuito que utiliza a rota que passa pelos três enlaces e que demanda três *slots* de frequência, deve-e, inicialmente, procurar por três *slots* livres adjacentes em cada um dos enlaces para garantir a contiguidade do espectro. Na Figura 1, há a existência de três *slots* livres adjacentes em cada enlace, são eles: *slots* 3 a 8 no enlace 1, *slots* 1 a 4 e 7 a 10 no enlace 2 e a 2 a 9 no enlace 3 .

O próximo passo é procurar a mesma faixa de espectro livre em cada enlace. Ou seja, devem existir os mesmos três *slots* livres simultâneamente em todos os enlaces. Se

os *slots* 1, 2 e 3 forem selecionados no enlace 1, perceba que o *slot* 5 não estará disponível no enlace 2. Apesar do mesmo *slot* está disponível no enlace 3, por conta da situação do enlace 2 o circuito não será estabelecido. Desta forma, no exemplo da Figura 1 não há a disponibilidade de três *slots* livres para respeitar a restrição da continuidade. Por conta disso a requisição de estabelecimento do circuito sofrerá bloqueio por fragmentação [Chatterjee et al. 2015]. O bloqueio por fragmentação ocorrerá sempre que as propriedades de continuidade e/ou contiguidade forem desrespeitadas.

Chama-se de fragmentação vertical a fragmentação que impede o estabelecimento de circuitos devido à restrição de contiguidade de espectro. Já a fragmentação que impede o estabelecimento de circuitos devido à restrição de continuidade de espectro é denominada de fragmentação horizontal [Talebi et al. 2014].

A arquitetura tradicional de uma rede óptica elástica é formada, basicamente, de BVT (*Bandwidth Variable Transpoder*) e BV-WXC (*Bandwidth Variable Cross-connect*). O BVT é um transponder capaz de adaptar a largura de banda através do ajuste da taxa de bits a ser transmitido ou do formato da modulação. A modulação em uma rede óptica elástica é um fator relevante. Para transmissões de longas distâncias são utilizadas modulações que alcancem maiores distâncias. Porém, modulações dessa natureza são menos eficientes em termos de alocação do espectro, ou seja, necessitam alocar mais *slots*.

As modulações QPSK (*Quadrature Phase-Shift Keying*) ou BPSK (*Binary Phase-Shift Keying)*, são exemplos de formatos de modulação com essas características. Modulações como o 32-QAM e 64-QAM transportam mais símbolos por bits do que a QPSK e BPSK. A consequência disso é que modulações dessa natureza demandam menos *slots* alocados, porém alcançam distâncias menores. Com isso, um BVT deve ser capaz de selecionar o melhor tipo de modulação a ser utilizada ponderando distância e uso eficiente de espectros.

3. Tolerância a Falhas em Redes Ópticas Elásticas

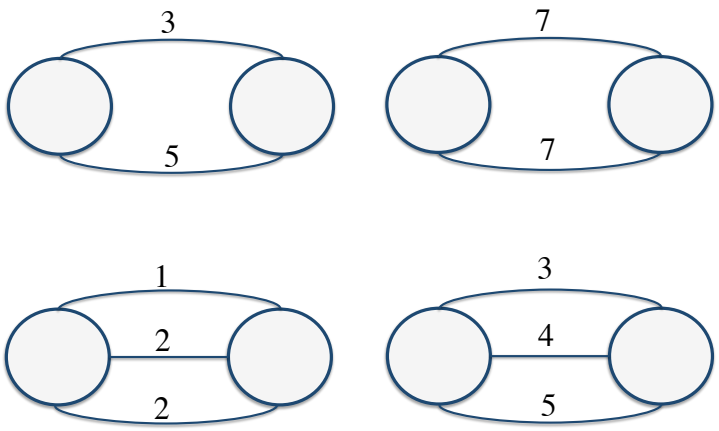
Nos últimos anos, várias técnicas têm sido propostas para tratar tolerância a falha em redes ópticas elásticas [Wang et al. 2015b], [Wei et al. 2014], [Amar et al. 2015] [Chen et al. 2015], [Shen et al. 2014]. Em geral, o objetivo é diminuir a probabilidade de bloqueio da rede e ao mesmo tempo conseguir garantir um nível de disponibilidade próximo a 100%.

Em [Oliveira and d. Fonseca 2014], são apresentados dois algoritmos para prover proteção de caminho através de uma técnica denominada *p-cycle*, que é uma estratégia que forma estruturas em anel para prover proteção. Tais algoritmos foram capazes de garantir 100% de proteção na ocorrência de falha simples, ou de até duas falhas simultâneas. Outros trabalhos também fazem uso de *p-cycle* para prover tolerância a falhas em redes ópticas elásticas como em [Wu et al. 2014] e [Ji et al. 2013].

Em [Wang et al. 2015a], os autores apresentam um algoritmo RSA tolerante a falha simples, que faz uso de proteção compartilhada de caminho. Neste algoritmo, é usado o conceito de SWP (*Spectrum Window Plane*), em que serão criados vários planos contendo subtopologias da topologia original. Desta forma o algoritmo pode tentar encontrar a melhor solução para cada plano, de forma a achar, ao final de várias iterações, uma rota

e um grupo de *slots* livres tanto para a rota principal, quanto para a rota de *backup*. Este algoritmo também escolhe modulações distintas para as duas rotas, visto que nem sempre a mesma modulação é compatível para as duas rotas encontradas, devido ao fato de suas distâncias serem diferentes.

Em [Ruan and Zheng 2014], os autores propõe um algoritmo baseado no conceito de *multipath*, que será referenciado no restante deste artigo como SM-RSA (*Survivable Multipath Routing and Spectrum Allocation*). No SM-RSA, as informações vão trafegar em vários caminhos disjuntos e simultâneos até chegar ao seu destino. Desta forma, dada um requisição $r = o, d, B, q$, onde o é o nó de origem, d é o nó de destino, B é a largura de banda requisitada e q é o nível de tolerância a falha da rede, onde $0 \leq q \leq 1$. Para este algoritmo, um conjunto N ≥ 2 de rotas é definido. Dada cada rota N, é necessário que cada uma dessas rotas tenha um nível de tolerância a falhas definido por q. A Figura 2 representa um exemplo de duas requisições com a mesma demanda ($B = 10$).



(a) *B*=10; *q*=0,3 (b) *B*=10; *q*=0,7

Figura 2. Exemplo de dois tipos de demandas, com a mesma quantidade de largura de banda requisitada e diferentes níveis de proteção garantidas pelo algoritmo SM-RSA

Na Figura 2 (a), temos o nível de tolerância a falhas $q = 0,3$ e um exemplo com dois caminhos ($N = 2$) e outro com três caminhos ($N = 3$). Para garantir o nível de tolerância a falhas requerido de $q = 0,3$, devemos garantir que com a falha de quaisquer caminhos de N, a soma dos caminhos restantes sejam maior ou igual a $B*q$ ($10*0,3 = 3$). Desta forma, a Figura 2 (a) com N=2 (dois caminhos) consegue esta garantia visto que caso ocorra uma falha no primeiro caminho, o caminho restante está acima do nível de tolerância a falhas ($5 \geq 3$). Neste mesmo exemplo, caso ocorra uma falha no segundo caminho, também garantiríamos a propriedade ($3 \geq 3$). Os dois exemplos da Figura 2 (b) também estão obedecendo a propriedade, visto que na retirada de qualquer um dos enlaces, a soma dos restantes é maior que $B * q$ ($10 * 0,7 = 7$).

A solução do algoritmo SM-RSA garante um nível de disponibilidade da rede de até 100% e mostra-se eficiente se comparada a um cenário de caminho simples [Ruan and Zheng 2014]. Entretanto, não é avaliado o seu comportamento em termos de

probabilidade de bloqueio, em um cenário ciente de imperfeições de camada física. Este trabalho analisa o impacto da camada física no algoritmo SM-RSA, observando assim a eficiência da técnica em um contexto mais próximo da realidade.

4. Modelo de Camada Física

O sinal óptico sofre degradação de qualidade ao ser transmitido, tanto nos dispositivos dos nós da rede, quanto nos próprios enlaces. Esta degradação pode ser dividida em duas classes: Efeitos Lineares (LI – *Linear Impairments*) e Efeitos Não Lineares (NLI – *Nonlinear Impairments*) [Rahbar 2012]. Os Efeitos Lineares são aqueles independentes da potência do sinal, como a Dispersão Cromática (CD - *Chromatic Dispersion*), a Emissão Espontânea Amplificada (ASE - *Amplified Spontaneous Emission*) e a Atenuação da Fibra. Já os Efeitos Não Lineares são dependentes da potência dos sinais ópticos e podem causar interferência tanto no próprio circuito, como nos seus vizinhos. A Auto-Modulação de Fase (SPM - *Self-Phase Modulation*), a Modulação de Fase Cruzada (XPM - *Cross-Phase Modulation*) e Mistura de Quatro Ondas (FWM - *Four-Wave Mixing*) são exemplos de efeitos não lineares.

Na região linear do OFDM óptico, o alcance de transmissão é limitado pelo ruído ASE, enquanto na região não linear ele é limitado pelos efeitos não lineares da fibra tais como o FWM, XPM e SPM [Beyranvand and Salehi 2013]. Os ruídos ASE e NLI são considerados neste trabalho pela adoção do modelo de camada física proposto em [Johannisson and Agrell 2014, Zhao et al. 2015].

Em uma rede óptica elástica, tais efeitos de camada física podem impactar na qualidade do sinal óptico, pois a sua taxa de erro de *bit* (BER – *Bit Error Rate*) pode se tornar intolerável. Neste sentido, se a BER chegar a níveis elevados, a qualidade de transmissão (QoT – *Quality of Transmission*) será impactada, o que poderá gerar um bloqueio por QoT [Beyranvand and Salehi 2013]. Os receptores ópticos possuem uma curva de desempenho que relaciona a SNR (*Signal to Noise Ratio*) diretamente com a BER, portanto, a SNR pode ser usada como critério de QoT de camada física de um circuito óptico.

O cálculo da SNR para um circuito i usando uma rota r_i é expresso por:

$$SNR_i = \frac{I}{I_{ASE} + I_{NLI}}. \tag{1}$$

A variável I é a densidade espectral da potência do sinal (PSD – *Power Spectral Density*), $I = P_{TX}/\Delta_f$, em que P_{TX} é potência de sinal e Δ_f é a largura de banda do circuito. A PSD do ruído ASE é dada por:

$$I_{ASE} = \sum_{l \in r_i} N_l I^0_{ASE}, \tag{2}$$

em que N_l é o número de spans do enlace l e $I^0_{ASE} = (G_{AMP} - 1)Fhv$. A variável F é o fator de emissão espontânea, que corresponde à metade da figura de ruído (NF – *Noise Figure*) do amplificador [Beyranvand and Salehi 2013], h é a constante de Planck, v é a frequência da luz e G_{AMP} é o ganho do amplificador óptico. A PSD do ruído dos efeitos não lineares (NLI – *Nonlinear Impairments*) é dada por:

$$I_{NLI} = \sum_{l \in r_i} N_l I^l_{NLI}, \tag{3}$$

em que I^l_{NLI}, a PSD do ruído NLI em um único span do enlace l, é expressa pela Equação 4 [Zhao et al. 2015]:

$$I^l_{NLI} = \frac{3\gamma^2 I^3}{2\pi\alpha|\beta_2|}\left(asinh\left(\frac{\pi^2|\beta_2|}{2\alpha}B_i^2\right) + \sum_j \ln\left[\left(\Delta_{f_{ij}} + \frac{B_j}{2}\right) / \left(\Delta_{f_{ij}} - \frac{B_j}{2}\right)\right]\right), \quad (4)$$

em que, j é outro circuito usando o enlace l, Bi e Bj são, respectivemente, as larguras de bandas para os circuitos i e j, $\Delta_{f_{ij}}$ é o espaçamento da frequência central entre os circuitos i e j, γ é o coeficiente não linear da fibra, β_2 é o parâmetro de dispersão da fibra e α é a atenuação de potência causada pela fibra.

Caso os níveis de QoT não sejam adequados, a requisição pode ser bloqueada por QoTN ou QoTO [Fontinele et al. 2016]. O QoTN é o bloqueio sofrido caso a nova requisição não atinja os níveis adequados de QoT. Mesmo que uma nova requisição atinja tal requisito, ela ainda poderá sofrer bloqueio caso o estabelecimento da nova requisição impacte na QoT dos circuitos já estabelecidos, ocasionando assim o QoTO. Estes dois tipos de bloqueio serão usados na avaliação de desempenho deste trabalho.

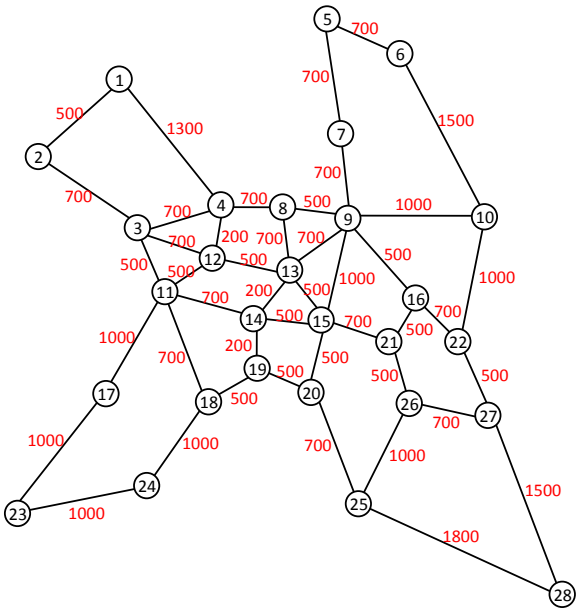
5. Avaliação de Desempenho

Para avaliação de desempenho, o algoritmo SM-RSA proposto em [Ruan and Zheng 2014] foi analisado em termos de probabilidade de bloqueio de circuitos em quatro cenários. O primeiro e o segundo cenários usam a topologia NSFNet (14 nós), sendo que no primeiro cenário será negligenciado o impacto da degradação dos efeitos da camada física e no segundo não. Já no terceiro e quarto cenário, a topologia usada será a EON (28 nós), sendo o terceiro sem considerar os impactos dos efeitos da camada física e o quarto com os impactos dos efeitos da camada física.

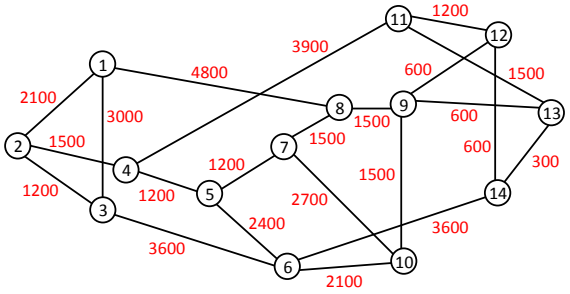
Nos cenários sem considerar os impactos dos efeitos da camada física, a escolha do formato de modulação para uma dada requisição de circuito é feita pelo seu alcance máximo de transmissão, como na maioria dos trabalhos apresentados na literatura [Gong et al. 2013, Talebi et al. 2014, Oliveira and d. Fonseca 2014, Wang et al. 2015c, Ruan and Zheng 2014, Chatterjee et al. 2015]. Já nos cenários que consideram os impactos dos efeitos de camada física, a QoT alcançada para cada formato de modulação é levada em conta para determinar qual deve ser o formato de modulação adequado. Essa estratégia é a mesma aplicado em [Beyranvand and Salehi 2013]. Onde, para uma dada requisição é aplicado cada formato de modulação para a rota seleciona e verifica-se a QoT alcançada para cada formato de modulação. Escolhe-se o formato de modulação com maior eficiência espectral que alcança uma QoT aceitável (SNR acima ou igual ao limiar de SNR exigido pelo formato de modulação).

A técnica de avaliação de desempenho adotada foi a simulação computacional. Foi utilizado o simulador SNetS (*SLICE Network Simulator*) [Santos 2015]. O SNetS é uma ferramenta de simulação desenvolvida para permitir a avaliação de desempenho de redes ópticas elásticas OFDM [Santos 2015]. Informações sobre a validação da ferramenta podem ser encontradas em [Santos 2015].

A Figura 3 mostra as duas topologias usadas neste estudo. O valor apresentado em cada enlace representa a distância do enlace em quilômetros.



(a) EON, 28 nós.



(b) NSFNet, 14 nós.

Figura 3. Topologias usadas no estudo de avaliação de desempenho. O número em cada enlace corresponde a distância em km.

Foram geradas 100000 requisições de circuitos em cada simulação. A geração de requisições é um processo de Poisson com taxa média de λ e o tempo médio de retenção dos circuitos é distribuído exponencialmente com média $1/\mu$. A carga de tráfego é distribuída uniformemente entre todos os pares de nós origem e destino. A carga em Erlangs pode ser definida por $\rho = \lambda/\mu$. Para cada simulação foram realizadas 10 replicações com diferentes sementes de geração de variável aleatória. Todos os resultados possuem nível de confiança de 95%.

Para o algoritmo SM-RSA, o parâmetro q foi definido por $q = 1$, o que significa um nível de proteção de 100%. O valor de N foi definido como $N = 2$, que significa que para cada requisição serão usados dois caminhos disjuntos para o tráfego das informações. Os requisitos de taxas de *bits* para cada circuito requisitado variam uniformemente entre 10 Gbps, 40 Gbps, 80 Gbps, 100 Gbps, 160 Gbps, 200 Gbps e 400 Gbps. Os formatos de modulação considerados neste estudo foram BPSK, QPSK, 8QAM, 16QAM, 32QAM e 64QAM [Gong et al. 2013]. Todos os enlaces da rede são bidirecionais e possuem largura de banda do espectro dividida em 400 *slots* de frequência [Wang et al. 2015a], [Horota et al. 2014], sendo que cada *slot* de frequência possui largura de banda de 12,5

GHz [Chatterjee et al. 2015]. Outros parâmetros utilizados nas simulações estão listados na Tabela 1 [Beyranvand and Salehi 2013, Zhao et al. 2015]. Os limiares de SNR para cada formato de modulação são 6 dB (BPSK), 9 dB (QPSK), 12 dB (8QAM), 15 dB (16QAM), 18 dB (32QAM) e 21 dB (64QAM) [Beyranvand and Salehi 2013].

Tabela 1. Parâmetros de camada física utilizados nas simulações.

Descrição	Valor
Densidade espectral de potência do sinal	-17 dBm/GHz
Atenuação da fibra	0,2 dB/km
Parâmetro de dispersão da fibra	16 ps^2/km
Coeficiente não linear da fibra	1,3 $(Wkm)^{-1}$
Tamanho de um span	100 km

A Figura 4 mostra a probabilidade de bloqueio de circuito em função da carga da rede (em erlangs) para o primeiro e o segundo cenário. A nomenclatura SM_sem_CF refere-se ao algoritmo SM-RSA sem as restrições de camada física, enquanto SM_com_CF refere-se ao algoritmo SM-RSA com as restrições de camada física. Considera-se que uma rede para se tornar praticável operacionalmente deverá possuir no máximo 20% de bloqueio. Ao comparar as duas curvas, podemos observar um aumento expressivo da probabilidade de bloqueio em todos os pontos de carga. No quarto ponto, por exemplo, há um aumento de 77%. Dessa forma, ao inserir camada física a probabilidade de bloqueio cresce de tal forma que a rede torna-se inviável (probabilidade de bloqueio acima de 20%).

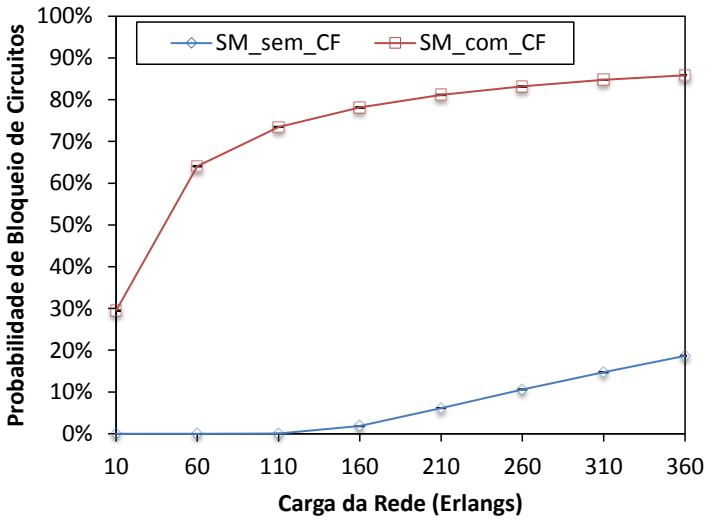


Figura 4. Valores de probabilidade de bloqueio em relação a carga da rede para a topologia NSFNet.

A Figura 5 apresenta a composição da probabilidade de bloqueio geral apresentada no cenário com restrições de camada física (segundo cenário). Observa-se que no último ponto de carga (360 erlangs), de um total de 86% de bloqueio, aproximadamente 45% foi causado devido a bloqueio de QoTN, aproximadamente 41% devido a bloqueio de QoT nos outros (QoTO) e menos de 1% devido ao bloqueio de fragmentação. Isto evidência o peso das restrições de camada física na probabilidade de bloqueio geral da rede.

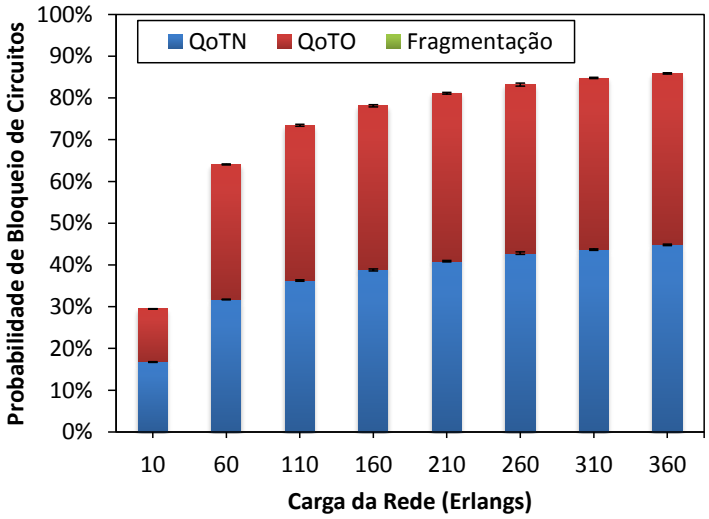


Figura 5. Composição da probabilidade de bloqueio geral para a topologia NSF-Net. Destacando os bloqueios QoTN, QoTO e por Fragmentação

A Figura 6 mostra a probabilidade de bloqueio de circuito para a topologia EON, terceiro e quarto cenário. Podemos observar pela Figura 6 um aumento de máximo aproximado de 75%, na carda de 250 Erlangs, quando consideramos um cenário que apresenta restrições de camada física. No cenário sem as restrições de camada física a grande maioria dos bloqueios ocorre devido à fragmentação do espectro. E a menor parte dos bloqueios ocorre por não existir espectro livre de forma alguma para o estabelecimento do circuito óptico requisitado.

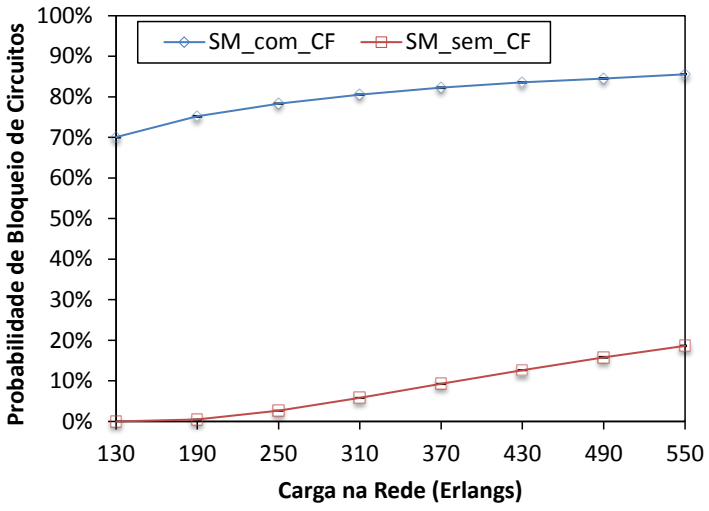


Figura 6. Valores de probabilidade de bloqueio em relação a carga da rede para a topologia EON.

A Figura 7 apresenta a composição da probabilidade de bloqueio geral apresentada no cenário com restrições de camada física, quarto cenário. Observa-se que no último ponto de carga (550 erlangs), de um total de 86% de bloqueio, aproximadamente 34% foi causado devido a bloqueio de QoTN, 52 % devido a bloqueio de QoTO e menos de

1% devido ao bloqueio de fragmentação, o que também evidencia o peso das restrições de camada física na probabilidade de bloqueio geral da rede. Assim como ocorreu no segundo cenário, o bloqueio por fragmentação é bem inferior aos bloqueios por QoTN e QoTO. Isto acontece porque os bloqueios por QoTN e QoTO levam a uma baixa utilização da rede, gerando pouca utilização do espectro no enlaces.

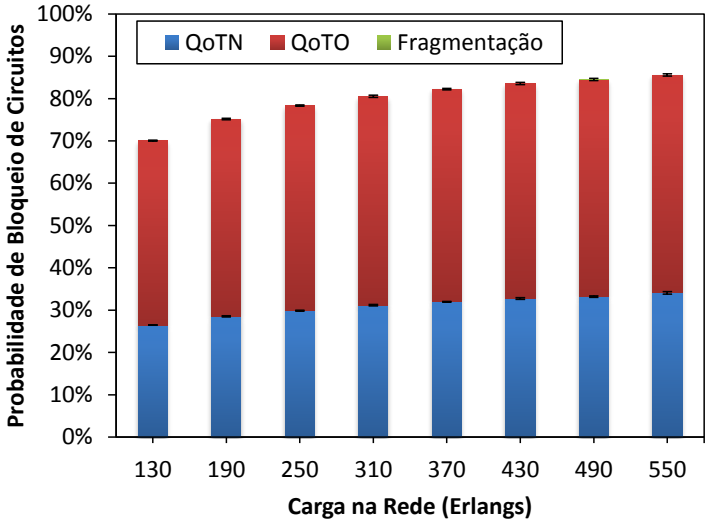


Figura 7. Composição da probabilidade de bloqueio geral para a topologia EON. Destacando os bloqueios QoTN, QoTO e por Fragmentação

Ao mudar a topologia de rede, observa-se uma inversão no tipo de bloqueio mais impactante. Na topologia NSFNet, temos o bloqueio QoTN sendo o que mais impacta na probabilidade de bloqueio geral, com 47% no ultimo ponto de carga. Já na topologia EON, o bloqueio QoTO aparece como o que mais influencia no desempenho da rede, chegando a taxas de 52%. O tamanho médio das rotas na topologia EON é maior que o tamanho médio das rotas na topologia NSFNet. Circuitos ópticos que tendem a possuir rotas com muitos saltos (ou que percorrem grandes distâncias) são mais propícios a possuírem uma maior probabilidade de bloqueio geral (comporta por fragmentação, QoTN, QoTO e por ausência de espectro livre). Quando circuitos com rotas com muitos saltos são estabelecidos são geradas degradações em uma quantidade maior de circuitos, em comparação a circuitos com rotas com poucos saltos. O estabelecimento de circuitos com rotas com muitos saltos pode levar a uma maior probabilidade de bloqueio por QoTO.

6. Conclusões

A principal contribuição deste trabalho é evidenciar o impacto que a degradação de camada física causa no desempenho dos algoritmos RSA tolerante a falhas. Isso porque a grande maioria dos trabalhos relacionados acabam por deixar de lado este cenário. Como as limitações de camada física são inerentes à tecnologia de redes ópticas, é crucial que elas sejam levadas em consideração para que a avaliação de desempenho seja realizada de forma mais fidedigna.

Observou-se que nas duas topologias analisadas, quando inserida as restrições de camada física, a probabilidade de bloqueio teve um aumento considerável. Tal aumento, devido aos bloqueios de QoT, chegou a aproximadamente 77% quando submetido

a 160 erlangs para a topologia NSFNet e aproximadamente 76% quando submetido a 250 erlangs na topologia EON. O que reforça a necessidade de considerar as degrações de camada física no planejamento de redes ópticas elásticas tolerante a falhas.

Para trabalhos futuros vamos analisar o comportamento do algoritmo SM-RSA em cenários que utilize um maior número de caminhos, além de diferentes níveis de tolerância a falhas. Serão avaliados outros algoritmos RSA tolerantes a falha em cenário ciente de camada física, para que se possa identificar de forma mais concisa o impacto causado pelas restrições de camada física. Além de propor um novo algoritmo RSA que garanta disponibilidade e sofra menos impacto das restrições de camada física, para que seja garantida baixas taxas de probabilidade de bloqueio na rede.

Referências

Amar, D., Rouzic, E. L., Brochier, N., and Lepers, C. (2015). Multilayer restoration in elastic optical networks. In *2015 International Conference on Optical Network Design and Modeling (ONDM)*, pages 239–244.

Beyranvand, H. and Salehi, J. (2013). A quality-of-transmission aware dynamic routing and spectrum assignment scheme for future elastic optical networks. *Journal of Lightwave Technology*, 31(18):3043–3054.

Chatterjee, B., Sarma, N., and Oki, E. (2015). Routing and spectrum allocation in elastic optical networks: A tutorial. *IEEE Communications Surveys Tutorials*, 17(3):1776–1800.

Chen, X., Zhu, S., Chen, D., Hu, S., Li, C., and Zhu, Z. (2015). On efficient protection design for dynamic multipath provisioning in elastic optical networks. In *2015 International Conference on Optical Network Design and Modeling (ONDM)*, pages 251–256.

Christodoulopoulos, K., Tomkos, I., and Varvarigos, E. A. (2011). Elastic bandwidth allocation in flexible ofdm-based optical networks. *Journal of Lightwave Technology*, 29(9):1354–1366.

Fontinele, A., Santos, I., Durães, G., and Soares, A. (2016). Achievement of fair and efficient regenerator allocations in translucent optical networks using the novel regenerator assignment algorithm. *Optical Switching and Networking*, 19, Part 1:22 – 39.

Gong, L., Zhou, X., Liu, X., Zhao, W., Lu, W., and Zhu, Z. (2013). Efficient resource allocation for all-optical multicasting over spectrum-sliced elastic optical networks. *IEEE/OSA Journal of Optical Communications and Networking*, 5(8):836–847.

Horota, A., Figueiredo, G., and Fonseca, N. (2014). Algoritmo de roteamento e atribuição de espectro com minimização de fragmentação em redes ópticas elásticas. *XXXII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*.

Ji, F., Chen, X., Lu, W., Rodrigues, J. J. P. C., and Zhu, Z. (2013). Dynamic p-cycle configuration in spectrum-sliced elastic optical networks. In *2013 IEEE Global Communications Conference (GLOBECOM)*, pages 2170–2175.

Jinno, M., Takara, H., Kozicki, B., Tsukishima, Y., Sone, Y., and Matsuoka, S. (2009). Spectrum-efficient and scalable elastic optical path network: architecture, benefits, and enabling technologies. *IEEE Communications Magazine*, 47(11):66–73.

Johannisson, P. and Agrell, E. (2014). Modeling of nonlinear signal distortion in fiber-optic networks. *Journal of Lightwave Technology*, 32(23):4544–4552.

Oliveira, H. M. N. S. and d. Fonseca, N. L. S. (2014). Protection in elastic optical networks against up to two failures based fipp p-cycle. In *2014 Brazilian Symposium on Computer Networks and Distributed Systems (SBRC)*, pages 369–375.

Rahbar, A. G. (2012). Review of dynamic impairment-aware routing and wavelength assignment techniques in all-optical wavelength-routed networks. *IEEE Communications Surveys Tutorials*, 14(4):1065–1089.

Ramaswami, R. and Sivarajan, K. N. (2009). *Optical Network - A Practical Perspective*. Morgan Kaufmann Publishers, 3th edition.

Ruan, L. and Zheng, Y. (2014). Dynamic survivable multipath routing and spectrum allocation in ofdm-based flexible optical networks. *IEEE/OSA Journal of Optical Communications and Networking*, 6(1):77–85.

Santos, I. (2015). *Alocação de Recursos para o Estabelecimento de Circuitos em Redes Ópticas WDM e OFDM*. Universidade Federal do Piauí, Teresina.

Shen, G., Wei, Y., and Bose, S. K. (2014). Optimal design for shared backup path protected elastic optical networks under single-link failure. *IEEE/OSA Journal of Optical Communications and Networking*, 6(7):649–659.

Talebi, S., Alam, F., Katib, I., Khamis, M., Salama, R., and Rouskas, G. N. (2014). Spectrum management techniques for elastic optical networks: A survey. *Optical Switching and Networking*, 13:34 – 48.

Wang, C., Shen, G., and Bose, S. K. (2015a). Distance adaptive dynamic routing and spectrum allocation in elastic optical networks with shared backup path protection. *Journal of Lightwave Technology*, 33(14):2955–2964.

Wang, C., Shen, G., Chen, B., and Peng, L. (2015b). Protection path-based hitless spectrum defragmentation in elastic optical networks: Shared backup path protection. In *2015 Optical Fiber Communications Conference and Exhibition (OFC)*, pages 1–3.

Wang, X., Brandt-Pearce, M., and Subramaniam, S. (2015c). Impact of wavelength and modulation conversion on translucent elastic optical networks using milp. *J. Opt. Commun. Netw.*, 7(7):644–655.

Wei, Y., Shen, G., and Bose, S. K. (2014). Span-restorable elastic optical networks under different spectrum conversion capabilities. *IEEE Transactions on Reliability*, 63(2):401–411.

Wu, J., Liu, Y., Yu, C., and Wu, Y. (2014). Survivable routing and spectrum allocation algorithm based on p-cycle protection in elastic optical networks. *Optik - International Journal for Light and Electron Optics*, 125(16):4446 – 4451.

Zhao, J., Wymeersch, H., and Agrell, E. (2015). Nonlinear impairment-aware static resource allocation in elastic optical networks. *Journal of Lightwave Technology*, 33(22):4554–4564.

Utilização da Árvore SPQR para um Cálculo Mais Eficiente das Medidas de Conectividade Baseadas em Cortes de Vértices

Henrique Hepp[1], **Jaime Cohen**[2], **Elias P. Duarte Jr.**[1]

[1]Departamento de Informática – Universidade Federal do Paraná (UFPR)
Caixa Postal 19018 – CEP 81531-980 – Curitiba – PR – Brazil.

[2]Departamento de Informática – Universidade Estadual de Ponta Grossa (UEPG)
Av. General Carlos Cavalcanti 4748 – Bloco L - Sala 103 CEP 84030-900 – PR – Brazil.

hhepp@inf.ufrpr.br, jaimecohen@gmail.com, elias@inf.ufpr.br

Abstract. *The connectivity measure $\kappa_2(v)$ of a vertex v of graph G is the maximum number of vertex-disjoint paths between v and any other vertex of the graph. The purpose of this work is to improve the efficiency for computing $\kappa_2(v)$. A pre-processing step is proposed in which a SPQR tree is buit that identifies all tri-connected components of the graph. Thus, $\kappa_2(v)$ can be computed separately for each component. The proposal was implemented and tested in graphs that model concrete cases. Experimental results are presented showing that for graphs for which the proposed strategy does improve the efficiency for computing $\kappa_2(v)$, in comparison with the original approach.*

Resumo. *A medida de conectividade $\kappa_2(v)$ de um vértice v em um grafo G é o número máximo de caminhos vértices disjuntos que há entre esse vértice e qualquer outro vértice desse grafo. A proposta desse trabalho é melhorar a eficiência do cálculo de $\kappa_2(v)$ através de um pré-processamento do grafo. Nessa etapa, é construída uma árvore SPQR que identifica as componentes triconexas do grafo. Dessa forma, o $\kappa_2(v)$ pode ser calculado separadamente para cada componente. A proposta foi implementada e testada em grafos que modelam casos concretos. Apresentamos resultados para grafos para os quais observou-se que o tempo de cálculo de com o pré-processamento é significativamente menor que o tempo para calcular $\kappa_2(v)$ no grafo original.*

1. Introdução

Nesse trabalho investigamos uma propriedade dos grafos que é a conectividade. No contexto desse trabalho, a conectividade é quantificada por meio de medidas que indicam o quão conectado está um vértice com o restante do grafo [Cohen et al. 2011, Pires et al. 2011]. Essas medidas indicam o quão resistentes são esses vértices para serem desconectados do grafo com a retirada de arestas ou vértices.

As medidas de conectividade utilizadas no trabalho são a 2-aresta-conectividade, $\lambda_2(v)$, e a 2-vértice-conectividade, $\kappa_2(v)$ propostas em [Cohen 2013]. Essas medidas correspondem ao número máximo de caminhos arestas/vértices disjuntos que há entre um vértice v e qualquer outro vértice desse grafo. O caminho é uma sequência de vértices em que de cada vértice há uma aresta para o próximo da sequência.

Essas medidas de conectividade podem ser aplicadas para tolerância a falhas em redes. Um exemplo é o caso de uma rede de computadores na qual deseja-se posicionar

vários servidores de acordo com a melhor configuração possível para que a comunicação entre dois dos servidores dificilmente seja interrompida devido a falhas da rede. Para tal, deve-se achar uma configuração em que os $\lambda_2(v)$ e $\kappa_2(v)$ desses servidores sejam os maiores possíveis.

A medida 2-vértice-conectividade tem um custo computacional alto, pois para calcular a medida é necessário calcular os menores cortes de vértices entre todos os pares de vértices do grafo. Um corte de vértices é um conjunto de vértices cuja remoção desconecta o grafo. O custo alto pode tornar impraticável o uso da medida de conectividade baseada em corte de vértices.

Para diminuir o custo, esse trabalho propõe melhorar a eficiência do cálculo da medida 2-vértice-conectividade através de um pré-processamento do grafo. O objetivo é separar o grafo em componentes vértice-biconexas e vértice-triconexas para que $\kappa_2(v)$ possa ser calculado separadamente para cada componente, que tem número de vértices menor que o do grafo original. Para fazer essa separação é usada a árvore SPQR, descrita a seguir.

A árvore SPQR [Gutwenger 2010] visa decompor o grafo em suas componentes vértice-triconexas. Essa árvore contém quatro tipos de nós: nó tipo S, onde os vértices formam um ciclo; nó tipo P, formado por dois vértices com três ou mais arestas em paralelo; nó tipo Q, o caso trivial com apenas uma aresta; e nó tipo R, uma componente triconexa que não corresponde aos outros casos.

Nesse trabalho, foi implementado um sistema que primeiro calcula as componentes biconexas de um grafo recebido como entrada. Em seguida, é executado um programa que recebe como entrada essas componentes e as transforma em árvores SPQR. No próximo passo, cada componente das árvores SPQR geradas é repassada para outro programa que calcula o $\kappa_2(v)$. Por último, como um vértice pode estar em mais de uma componente, os vértices que receberam mais de um $\kappa_2(v)$ ficam com o maior valor da medida de conectividade.

A proposta foi implementada e testada em grafos reais, como UsaAir97 [Batagelj and Mrvar 2006], o grafo que representa a rede de aeroportos dos EUA. Nos resultados obtidos observou-se que nos grafos utilizados existe uma componente biconexa principal, com a maior parte dos vértices, e, a partir dessa componente, pode ser obtida uma componente triconexa. Nesses casos, o tempo computacional para o cálculo de $\kappa_2(v)$ corresponde principalmente ao cálculo dessa componente triconexa, o que é significativamente menor, no caso de UsaAir97 foi de 68%, que o tempo para calcular $\kappa_2(v)$ no grafo original.

O restante deste trabalho está organizado da seguinte maneira. A seção 2 descreve as medidas de conectividade a seção 3 descreve informalmente a árvore SPQR e a sua aplicação no cálculo da medida $\kappa_2(v)$. A seção 4 descreve a implementação e os resultados experimentais. Por fim, a seção 5 conclui o trabalho.

2. Medidas de Conectividade

Existem diversas medidas para caracterizar o quão conectado está um vértice de um grafo [Nagamochi and Ibaraki 2008, Cohen 2013]. Uma das medidas de conectividade de vértices é o próprio grau. Em um grafo $G = (V, E)$, onde V é o conjunto de vértices e

E o conjunto de arestas, o *grau* de um vértice v é igual ao número de arestas incidentes a v e é denotado por grau(v). Quando o grau de um vértice tem um valor alto, geralmente ele está bem conectado com o grafo. Entretanto nem sempre é assim, por exemplo, considere o caso de um grafo que contém um subgrafo com topologia estrela. Veja a figura 1. O vértice 1, embora tenha um grau alto, será desconectado do grafo com a remoção do vértice 2. Nesse caso, um vértice tem um grau alto, mas é desconectado do grafo pela remoção de apenas um vértice ou aresta. Dessa forma, um grau alto não garante que um vértice esteja bem conectado.

Para termos mais informações sobre a conectividade de um vértice podemos usar outras medidas, como a medida de conectividade baseada em corte de arestas e a baseada em corte de vértices, descritas nas próximas seções.

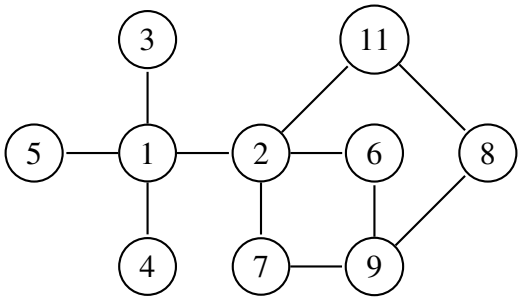


Figura 1. Grafo com o vértice 1 com topologia estrela, ele é desconectado do grafo retirando o vértice 2.

2.1. Medidas de Conectividade Baseadas em Cortes de Arestas

A medida de conectividade de vértices baseada em cortes de arestas [Duarte et al. 2004, Cohen et al. 2011] também chamada de i-aresta-conectividade é definida da seguinte forma:

Definição 1 *Seja $G = (V, E)$ um grafo não direcionado. Considere um conjunto de vértices $X \subseteq V, |X| \geq 2$. A aresta-conectividade de X em relação a G é o tamanho de um corte mínimo que separa quaisquer pares de vértices em X. Denotamos a aresta-conectividade de X por $\lambda(X)$. Se $|X| = 1$, com $X = \{v\}$, então definimos $\lambda(X) = grau(v)$.*

De acordo com essa definição, em grafos com arestas com capacidades unitárias, $\lambda(X)$ é um número que indica o menor número de arestas a serem retiradas de um grafo G para desconectar algum par de vértices de um conjunto X.

Definição 2 *A i-aresta-conectividade de um vértice v, denotada por $\lambda_i(v)$, é a máxima aresta-conectividade de um conjunto $X \subseteq V$ que satisfaz:*

i. $v \in X$, *e*
ii. $|X| \geq i$

Em particular, em grafos com capacidades unitárias, a medida $\lambda_2(v)$ de um vértice v é o maior número de caminhos aresta disjuntos entre v e qualquer outro vértice. A figura 2 mostra um exemplo. O vértice 1 tem 5 caminhos aresta disjuntos com o vértice 5, como o vértice 1 não tem um número maior de caminhos aresta disjuntos com qualquer outro vértice, $\lambda_2(1) = 5$.

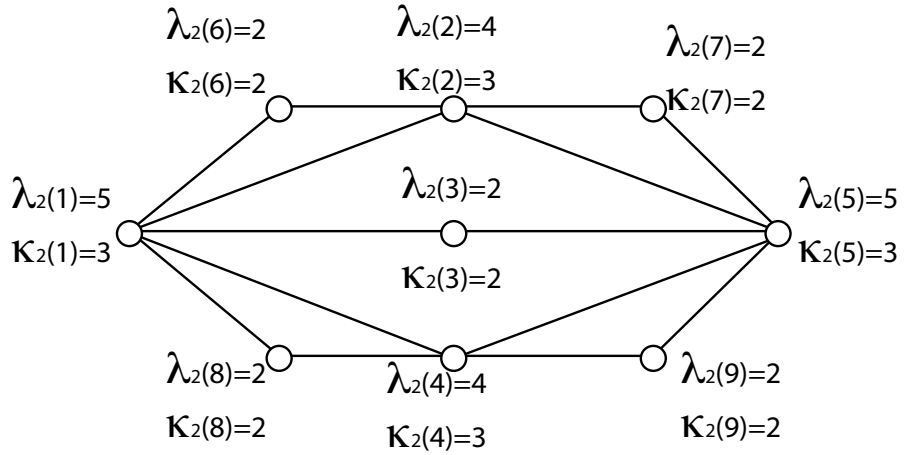


Figura 2. Grafo com os valores da 2-aresta-conectividade e da 2-vértice-conectividade [Pires et al. 2011, Pires 2011].

2.2. Medidas de Conectividade Baseadas em Cortes de Vértices

A medida de conectividade de vértices baseadas em corte de vértices, vértice-conectividade [Pires 2011, Pires et al. 2011] é definida a seguir:

Definição 3 *Dados dois vértices $s, t \in V$, a vértice-conectividade entre s e t, denotada por $\kappa(s,t)$, é o número de caminhos vértice disjuntos entre s e t*

Definição 4 *A vértice-conectividade de um conjunto $X \subseteq V$ é a menor vértice-conectividade entre quaisquer pares de vértices em X e é denotada por $\kappa(X)$.*

Definição 5 *Definimos a i-vértice-conectividade de um vértice v, denotada por $\kappa_i(v)$, como a maior vértice-conectividade de um conjunto $X \subseteq V$ satisfazendo:*

i. $v \in X$, *e*
ii. $|X| \geq i$

Em particular, a medida $\kappa_2(v)$ de um grafo G é o maior número de caminhos vértices disjuntos entre v e qualquer outro vértice de G. A figura 2 mostra um exemplo. O vértice 1 tem 3 caminhos vértice disjuntos com o vértice 5, como o vértice 1 não tem um número maior de caminhos vértice disjuntos com qualquer outro vértice, $\kappa_2(1) = 3$.

2.3. Algoritmo para Calcular a Medida de Conectividade Baseada em Cortes de Vértices

Para calcular a medida $\kappa_2(v)$ de cada vértice de um grafo $G = (V, E)$, o algoritmo proposto em [Pires et al. 2011, Pires 2011] requer o cálculo de $|V| - 1$ cortes mínimos entre pares de vértices. Para calcular esses cortes de vértices reduz-se o problema de corte de vértices ao corte de arestas. Essa redução é ilustrada nas figuras 3, 4 e 5.

No primeiro passo, mostrado na figura 3, o grafo não orientado é transformado em um grafo orientado. Para isso, as arestas do grafo original são duplicadas de modo que uma aresta $\{a, b\}$ corresponda a duas arestas direcionadas $\{(a, b), (b, a)\}$.

No segundo passo, na figura 4 é mostrado um subgrafo do grafo da figura 3, cada vértice v é duplicado para os vértices v' e v''. As arestas que tinham como origem o vértice

v, têm agora como origem o vértice v''. As arestas que tinham como destino o vértice v têm agora como destino o vértice v'. Por último, adiciona-se as arestas direcionadas (v', v'') entre todos os pares de vértices associados.

No terceiro passo, mostrado na figura 5, para garantir que o corte mínimo utilize arestas que correspondem ao grafo original atribui-se as seguintes capacidades às arestas: as arestas do tipo (v', v'') recebem capacidade de valor 1 e as demais arestas recebem capacidade $|V| - 1$.

Nesse grafo resultante é utilizado o algoritmo para calcular o fluxo máximo, que retorna o corte mínimo de arestas, que corresponde ao corte mínimo de vértices do grafo original.

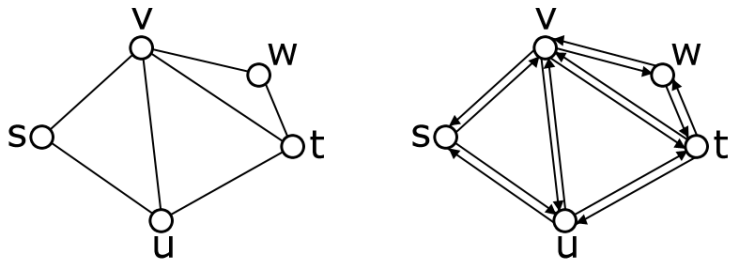


Figura 3. Transformação de grafo não orientado para orientado [Pires 2011].

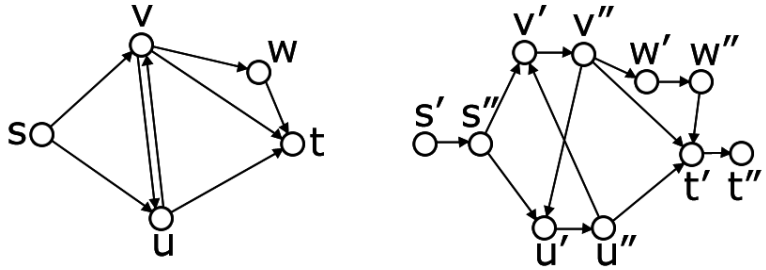


Figura 4. Transformação de grafo orientado para aplicação de corte de arestas [Nagamochi and Ibaraki 2008].

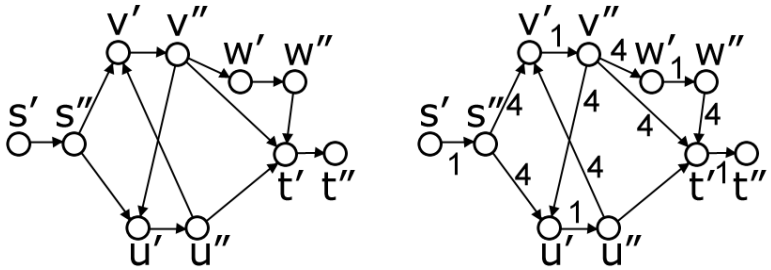


Figura 5. Aplicação de pesos para corte de arestas [Nagamochi and Ibaraki 2008].

2.3.1. Melhoria da Eficiência do Cálculo de $\kappa_i(v)$

Para reduzir o tempo de cálculo de $\kappa_i(v)$ podemos usar o seguinte lema [Pires et al. 2011, Pires 2011]:

Lema 1 *Dado um grafo $G = (V, E)$ e $v \in V$, tem-se que $\kappa_i(v) \leq \lambda_i(v)$ para todo $i \geq 2$.*

Esse lema parte da relação de que o número de caminhos vértice disjuntos entre dois vértices v e w é limitado pelo número de caminhos aresta disjuntos entre esses dois vértices.

Como o cálculo de $\lambda_i(v)$ é computacionalmente mais rápido, podemos usar esse fato no cálculo de $\kappa_i(v)$. De acordo com o lema 1, o $\lambda_i(v)$ é o limitante superior de $\kappa_i(v)$. Portanto, ao calcular os fluxos máximos entre v e os outros $|V| - 1$ vértices, o cálculo poderá ser interrompido no momento em que o número de caminhos vértice disjuntos alcançar o valor de $\lambda_i(v)$.

3. Utilização da Árvore SPQR no Cálculo de $\kappa_2(v)$

A árvore SPQR é uma árvore que contém as componentes vértice triconexas de um grafo vértice biconexo. Devido ao fato da definição e do algoritmo com tempo linear de construção dessa árvore, definidos formalmente em [Hopcroft and Tarjan 1972, Battista and Tamassia 1996, Gutwenger 2010], serem bastante complicados, não os apresentamos nesse artigo. Expomos apenas uma definição informal com as informações necessárias para a compreensão desse trabalho.

As árvores SPQR são construídas para grafos biconexos. Grafos que não sejam biconexos serão inicialmente decompostos em suas componentes biconexas. Dessa forma, assumiremos no restante desta seção que o grafo é biconexo.

A figura 6 apresenta um exemplo de uma árvore SPQR. A árvore SPQR é composta de quatro tipos de nós, os nós S, P, Q e R, definidos abaixo:

1. Nó tipo S. Corresponde ao caso dito serial. É um ciclo com três ou mais vértices. Um ciclo de três vértice é uma componente triconexa. Quando o ciclo tem mais de três vértices é uma componente biconexa.
2. Nó tipo P. Corresponde ao caso dito paralelo. É um grafo com dois vértices e 3 ou mais arestas entre eles.
3. Nó tipo Q. Corresponde ao caso dito trivial com apenas uma aresta. Várias implementações, como a usada nesse trabalho, omitem esse nó.
4. Nó tipo R. Corresponde ao caso dito rígido e é uma componente triconexa que não é do tipo S nem do tipo P.

Nas implementações mais recentes, as ligações entre os nós (S, P e R) são arestas virtuais, na figura 6 são as arestas pontilhadas. Essas arestas correspondem aos dois vértices que os dois nós compartilham.

Para esse trabalho, duas observações que decorrem da definição formal ainda são importantes:

1. Não existem dois nós vizinhos do mesmo tipo, pois nesses casos eles formariam apenas um nó.
2. Como pode-se ver pela figura 6, o nó P ocorre entre os nós S e R por existir uma aresta no grafo original entre os dois vértices do corte de vértices. Por outro lado, se houverem dois nós R e S vizinhos, então os dois vértices do corte de vértices não podem estar conectados por uma aresta no grafo original.

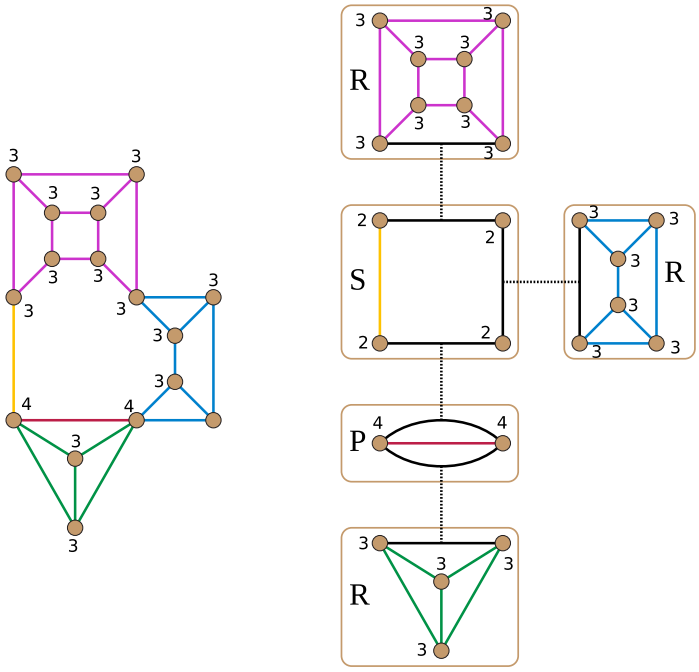


Figura 6. Exemplo de uma árvore SPQR com os valores de $\kappa_2(v)$.

3.1. Utilização da Árvore SPQR no Cálculo de $\kappa_2(v)$

Para melhorar a eficiência do cálculo $\kappa_2(v)$ para todo vértice de um grafo pode-se usar a árvore SPQR para identificar as componentes triconexas e depois calcular o $\kappa_2(v)$ separadamente para cada componente S, P, e R. Cada componente é separada do restante do grafo por cortes de dois vértices, enquanto que dentro da componente, os vértices podem ter entre eles múltiplos caminhos vértice disjuntos. Assim basta calcular o $\kappa_2(v)$ individualmente para cada componente.

Deve-se lembrar que há vértices que estão em mais de uma componente. O $\kappa_2(v)$ desses vértices pode ser calculado comparando-se entre eles o $\kappa_2(v)$ obtido em cada componente. O maior $\kappa_2(v)$ é o $\kappa_2(v)$ do grafo inteiro.

Observando as componentes de forma individual, podemos dizer que:

Componente S É o caso em que os vértices formam um ciclo, o $\kappa_2(v)$ para cada um desses vértices é, portanto, 2. Pois só há dois caminhos vértices disjuntos entre eles.

Componente R É uma componente triconexa que é repassada diretamente para o programa que faz o cálculo de $\kappa_2(v)$.

Componente P O $\kappa_2(v)$ dos vértices da componente P é o número de caminhos vértice disjuntos entre seus dois vértices no grafo original. Se formos comparar o grafo original com a árvore, veremos, por exemplo, pela figura 6, que entre os dois vértices de P existem 4 caminhos vértice disjuntos no grafo original. Já pela árvore, dentro da componente P existem apenas 3 caminhos vértice disjuntos. Isso mostra que para o cálculo de $\kappa_2(v)$ precisamos calcular o número de caminhos vértice disjuntos dos dois vértices de P com relação a todo grafo original, e não apenas dentro da componente P.

Depois de calcular o $\kappa_2(v)$ para cada componente, compara-se o $\kappa_2(v)$ dos vértices que estão em mais de uma componente. O maior $\kappa_2(v)$ entre eles é o $\kappa_2(v)$ do grafo. Veja por exemplo a figura 6, considere o vértice do canto inferior direito da componente S, $\kappa_2(v)$ = 2. Esse vértice também se encontra na componente R, canto inferior da esquerda com $\kappa_2(v)$ = 3, e na componente P, vértice da direita com $\kappa_2(v)$ = 4. O maior valor é o da componente P, por isso, o valor de $\kappa_2(v)$ desse vértice fica igual a 4.

4. A Implementação

Nesse trabalho, foi implementado um *script* em Python que primeiro calcula as componentes biconexas de um grafo recebido como entrada. Em seguida, o *script* chama um programa em Java que recebe como entrada essas componentes e as transforma em árvores SPQR. No próximo passo, cada componente das árvores SPQR gerada é repassada para outro programa, em C, que calcula o $\kappa_2(v)$. Por último, como um vértice pode estar em mais de uma componente, os vértices que receberam mais de um $\kappa_2(v)$ ficam com o maior valor da medida de conectividade. Dessa forma, a saída do *script* são os valores de $\kappa_2(v)$ do grafo recebido como entrada.

Os principais passos executados pelo *script* são os seguintes:

1. Recebe um grafo G como entrada
2. Calcula as componentes biconexas de G utilizando uma função da biblioteca NetworkX do Python
3. Para todo vértice v de componentes com apenas dois vértices, faz $\kappa_2(v) := 1$
4. Para cada componente biconexa, calcula a árvore SPQR usando a última versão (0.2.429) da biblioteca Java para grafos jBPT[1]
5. Para todo vértice v em componente S, faz $\kappa_2(v) := 2$
6. Para cada componente R, execute o programa em C escrito por Karine Pires[**?**, Pires 2011] que calcula $\kappa_2(v)$ para cada vértice v da componente
7. Para cada componente P, calcula o número de caminhos vértice disjuntos entre seus dois vértices usando o mesmo programa do passo anterior
8. Finalmente, compara o $\kappa_2(v)$ de cada vértice v existente em mais de uma componente, e faz $\kappa_2(v)_{final} :=$ o maior $\kappa_2(v)$

4.1. Resultados Experimentais

O pré-processamento de dados foi aplicado sobre grafos que modelam casos concretos: UsaAir97 [Batagelj and Mrvar 2006] representa os aeroportos dos EUA, Yeast [Bu et al. 2003] é uma rede de interações de proteínas, Rome [Storchi et al. 1999] representa as ruas de Roma, Geocomp2 [Batagelj and Mrvar 2006] e CA-GrQc [Kleinberg et al. 2007] são redes de colaboração científica e Powergrid [Watts and Strogatz 1998] é uma rede de distribuição de energia elétrica. Nas seções seguintes é apresentada uma síntese dos resultados.

4.1.1. Grafo: UsaAir97

Nessa seção são apresentados resultados experimentais obtidos para o grafo UsaAir97 que representa os aeroportos dos EUA. Foram feitas medidas para o cálculo de $\kappa_2(v)$

[1]Disponível em `https://code.google.com/p/jbpt/`

utilizando tanto o grafo original como dois tipos de pré-processamento: no primeiro, o cálculo é feito sobre as componentes biconexas, calculadas no passo dois do *script* acima, no segundo, o cálculo é feito utilizando as árvores SPQR.

Fazendo o pré-processamento do grafo UsaAir97, que tem 332 vértices e 2126 arestas, podemos observar que há, após passar a transformação em componentes biconexas, uma componente B com a maior parte dos vértices. Essa componente B tem o seu tamanho reduzido com a transformação da árvore SPQR para uma componente R. As demais componentes do grafo, sejam biconexas ou das árvores SPQR, têm um tamanho pequeno, de menos de 10 vértices. Essas componentes têm o $\kappa_2(v)$ calculado rapidamente.

Podemos visualizar esse fato na figura 7 e na tabela 1. De um grafo de 332 vértices, temos uma componente grande biconexa com 244 vértices, que são os vértices vermelhos junto com os vértices verdes. Os vértices em azul correspondem às demais componentes biconexas que para os quais o tempo de cálculo de $\kappa_2(v)$ é desprezível. Os vértices brancos formam a maior componente R, os vértices em verde correspondem às demais componentes da árvore SPQR que também têm tempos do cálculo dos $\kappa_2(v)$ desprezíveis. Fazendo o cálculo de $\kappa_2(v)$, observa-se que todos esses vértices para os quais os tempos de cálculo $\kappa_2(v)$ são desprezíveis têm valor de $\kappa_2(v)$ baixo (veja a figura 8). São os vértices com baixa conectividade no grafo. Desse modo, para o grafo da figura 7 observou-se que o pré-processamento foi efetivo. De acordo com a tabela 1, temos uma componente B com 73% do tamanho do grafo de entrada e uma componente R, da árvore SPQR, com 62% do tamanho do grafo de entrada.

O cálculo de $\kappa_2(v)$ foi feito em um computador com 8 GB de memória e com um processador Intel Core i7-3537U com 2.00 x 4. De acordo com a tabela 2, houve uma melhoria significativa no tempo de execução. Utilizando apenas componentes biconexas a melhoria é de 52% do tempo de execução, com a árvore SPQR a melhoria é de 68%.

Tabela 1. Tabela com o número de vértices de UsaAir97 após o pré-processamento. Ao lado está a porcentagem de vértices com relação aos vértices do grafo original.

Número de vértices	Grafo original	Maior componente B	Maior componente R
UsaAir97	332	244 73%	205 62%

Tabela 2. Tempo médio para 3 processamentos do cálculo de $\kappa_2(v)$ para todos os vértices do grafo UsaAir97.

Tempo grafo original	245s
Tempo componentes biconexas	117s 48% do original
Tempo árvore SPQR	79s 32% do original

4.1.2. Os Outros Grafos

Nessa seção são apresentados resultados experimentais obtidos para os grafos Yeast, Rome, Geocomp2, CA-GrQc e Powergrid. Como o número de vértices desses grafos

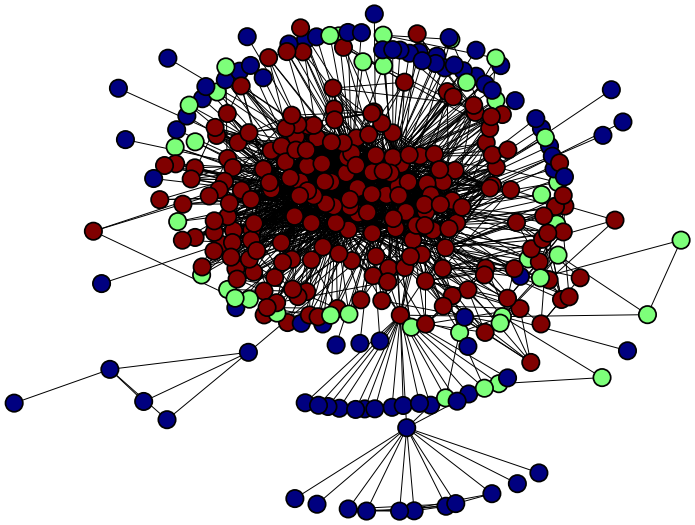


Figura 7. Grafo de UsaAir97, os vértices internos, em vermelho, correspondem ao maior componente R, a maior componente B corresponde aos vértices em vermelho junto com os verdes.

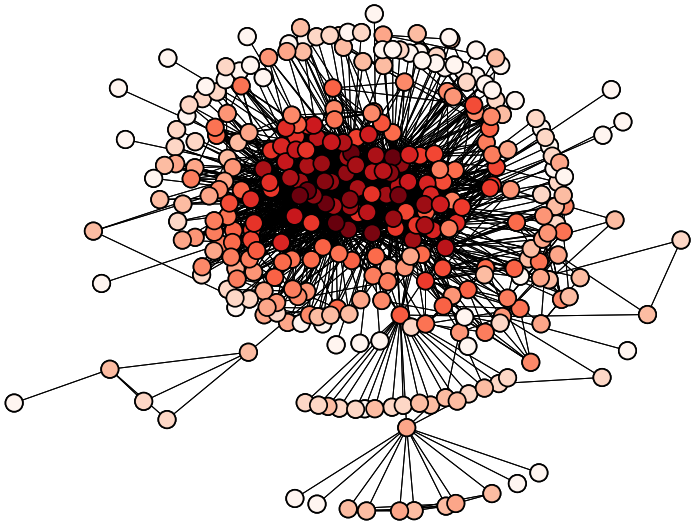


Figura 8. Grafo de UsaAir97, a tonalidade indica o valor de $\kappa_2(v)$, quanto mais escuro maior é o valor de $\kappa_2(v)$, os vértices em branco têm $\kappa_2(v) = 1$.

é maior que 1000, o tempo de processamento é muito grande (acima de 8 horas) para calcular o $\kappa_2(v)$. Por isso, foi apenas feita a construção da árvore SPQR e medido o tamanho das componentes obtidas.

Observou-se nesses grafos uma situação semelhante a observada no grafo UsaAir97, todos têm uma componente biconexa grande que é reduzida para uma componente R grande. Todas as demais componentes são pequenas e, portanto, têm tempos de cálculo de $\kappa_2(v)$ desprezíveis.

Pela tabela 3 pode-se verificar, de forma qualitativa, o potencial da redução de tempo de execução originada pelo pré-processamento. Com a árvore SPQR temos a menor redução de vértices para o grafo Rome, de apenas 33% com relação ao original. A maior redução foi obtida com o grafo Powergrid que foi de 79%.

Tabela 3. Tabela com o número de vértices após o pré-processamento. Ao lado está a porcentagem de vértices com relação aos vértices do grafo original.

Número de vértices	grafo original	maior componente B		maior componente R	
Yeast	2221	1464	66%	1137	51%
Rome	3353	2689	80%	2251	67%
Geocomp2	3621	1901	52%	1149	32%
CA-GrQc	4158	2651	64%	2456	59%
Powergrid	4941	3040	62%	1022	21%

5. Conclusão

Esse trabalho apresentou uma proposta para melhorar a eficiência do cálculo de $\kappa_2(v)$ através de um pré-processamento do grafo. Nessa etapa, o grafo é primeiro dividido em componentes biconexas e depois, para cada componente biconexa, é construída uma árvore SPQR que identifica as componentes triconexas do grafo. Dessa forma, o $\kappa_2(v)$ pode ser calculado separadamente para cada componente, que tem número de vértices menor que o do grafo original.

A proposta foi implementada e testada em grafos que modelam casos concretos. Nos resultados obtidos observou-se que nos grafos utilizados existe uma componente biconexa principal, com a maior parte dos vértices, e, a partir dessa componente, pode ser obtida uma componente triconexa tipo R. Para o grafo UsaAir97, com 332 vértices, que representa os aeroportos dos EUA obteve-se uma componente biconexa principal com 244 vértices e uma componente R com 205 vértices. Para o grafo Powergrid, com 4941 vértices, que representa uma rede de distribuição elétrica obteve-se uma componente biconexa com 3040 vértices e uma componente R com 1021 vértices. Nesses casos, o tempo computacional para o cálculo de $\kappa_2(v)$ corresponde principalmente ao cálculo de $\kappa_2(v)$ nessa componente triconexa tipo R, o que é significativamente menor que o tempo para calcular $\kappa_2(v)$ no grafo original. Para o grafo UsaAir97 obteve-se uma redução de 62% no tempo de cálculo de $\kappa_2(v)$ usando a árvore SPQR com relação ao cálculo de $\kappa_2(v)$ no grafo original.

Nesse trabalho, observou-se que a implementação da árvore SPQR pela biblioteca jBPT não é linear. Por isso, sugere-se para testes futuros uma biblioteca disponível em

C++ no site www.ogdf.net/doc-ogdf/. Um trabalho futuro a ser realizado é estudar o comportamento do algoritmo de cálculo do $\kappa_2(v)$ com a árvore SPQR diante da inserção ou remoção de vértices e arestas.

Referências

Cohen, J., Duarte, E. P. and Schroeder, J. (2011). Connectivity Criteria for Ranking Network Nodes, *Communications in Computer and Information Science (CCIS)*, Vol. 116. pp. 35-45.

Cohen, J. (2013). Algoritmos Paralelos para Árvores de Cortes e Medidas de Centralidade em Grafos, Tese de Doutorado, Departamento de Informática - Universidade Federal do Paraná,

Pires, K., Cohen, J. and Duarte, E. P. (2011). Medidas de Conectividade Baseadas em Cortes de Vertices para Redes Complexas, *12 Workshop de Testes e Tolerância a Falhas (WTF'2011), Anais do SBRC'2011*.

Pires, K. (2011). Medidas de Conectividade Baseadas em Cortes de Vértices para Redes Complexas, Dissertação de Mestrado, Departamento de Informática - Universidade Federal do Paraná.

Gutwenger, C. (2010). Application of SPQR-Trees in the Planarization Approach for Drawing Graphs, Tese de Doutorado, Technischen Universität Dortmund an der Fakultät für Informatik.

Batagelj, V. and Mrvar, A. (2006). Pajek datasets. Disponível em http://vlado.fmf.uni-lj.si/pub/networks/data/, Acessado em dezembro de 2015.

Nagamochi, H. and Ibaraki, T. (2008). *Algorithmic Aspects of Graph Connectivity*, Cambridge University Press.

Duarte, E. P., Santini, R. and Cohen, J. (2004). Delivering Packets During the Routing Convergence Latency Interval Through Highly Connected Detours, *DSN*, pp. 495–. IEEE Computer Society.

Hopcroft, J. E. and Tarjan, R. E. (1972). Finding the Triconected Components of a Graph, *Technical Report 72-140*, Cornell University.

Di Battista, G. and Tamassia, R. (1996). On-Line Maintenance of Triconnected Components with SPQR-Trees, *Algorithmica*, Vol. 15, pp. 302-318.

Bu, D., Zhao, Y., Cai, L., Xue, H., Zhu, X., Lu, H., Zhang, J., Sun, S., Ling, L., Zhang, N., Li, G., Chen, R. (2003). Topological Structure Analysis of the Protein-protein Interaction Network in Budding Yeast, *Nucleic acids research*, Vol. 31(9), pp. 2443–2450.

Storchi, G., Dell'Olmo, P. and Gentili, M. (1999) Directed Road Network of the City of Rome (1999). Disponível em http://www.dis.uniroma1.it/challenge9/download.shtml, acessado em dezembro de 2015.

Kleinberg, J. M., Leskovec J. and Faloutsos, C. (2007). Graph Evolution: Densification and Shrinking Diameters, *ACM Transactions on Knowledge Discovery from Data (ACM TKDD)*.

Watts, D. J. and Strogatz, S. H. (1998). Collective Dynamics of 'Small-World' Networks, *Nature*, Vol. 393(6684), pp. 440–442.

Avaliação do Controle de Acesso de Múltiplos Usuários a Múltiplos Arquivos em um Ambiente Hadoop

Eduardo Scuzziato[1], João E. Marynowski[1,2], Altair O. Santin[1]

[1]Escola Politécnica – Ciência da Computação
Pontifícia Universidade Católica do Paraná (PUCPR), Curitiba – PR – Brasil

[2]Setor de Educação Profissional e Tecnológica
Universidade Federal do Paraná (UFPR), Curitiba – PR – Brasil

scuzziato.eduardo@gmail.com, jeugenio@ufpr.br, santin@ppgia.pucpr.br

Abstract. *Massive processing of data is a reality for several computer systems. The security of processed data has great importance since the environment is typically shared among multiple users. This article presents an evaluation of the access control of multiple users and multiple files, considering the different control levels of a Hadoop environment (operating system, distributed file system and web interface). A test scenario is proposed and validated at different levels and different versions of a Hadoop distribution (Hortonworks). The versions presented the same behavior but we identified errors and differences between control levels.*

Resumo. *O processamento massivo de dados é uma realidade para diversos sistemas computacionais. A segurança dos dados processados é de grande importância, uma vez que o ambiente normalmente é compartilhado entre múltiplos usuários. Este artigo apresenta uma avaliação do controle de acesso de múltiplos usuários a múltiplos arquivos, considerando os diferentes níveis de controle de um ambiente Hadoop (sistema operacional, sistema de arquivo distribuído e interface web). Um cenário de teste é proposto e validado nos diferentes níveis e diferentes versões de uma distribuição do Hadoop (Hortonworks). As versões apresentaram mesmo comportamento mas identificamos erros e diferenças entre os níveis de controle.*

1. Introdução

Conjuntos de dados extremamente grandes são gerados e precisam ser manipulados diariamente excedendo a capacidade de processamento dos sistemas de banco de dados convencionais [Zikopoulos et. al. 2010]. Esses conjunto de dados são denominados big data e têm como características o grande volume, velocidade e a variabilidade dos dados [White 2012]. Hadoop é um framework de código aberto que permite o armazenamento e processamento distribuído de big data em um grande conjunto de máquinas (*cluster*) [Hadoop]. Nesse ambiente, o armazenamento é feito pelo HDFS (Hadoop Distributed File System) que é um sistema de arquivos distribuído escalável para grandes aplicações e com grande quantidade de dados distribuídos [HDFS]. O Hadoop também dispõe do Hue (Hadoop User Experience) [Hue], que é uma aplicação web que fornece aos seus usuários uma interface para a manipulação de outras ferramentas do ambiente Hadoop,

como o HDFS, Hive [Thusoo 2009] e Hbase [White 2012]; além da administração de usuários, que possibilita criar usuários e grupos.

Em um ambiente que possibilita diversas formas de uso e dispõe de grande poder de processamento como Hadoop, é natural a ideia de que ele possa ser compartilhado entre diferentes usuários e grupos [Tankard 2012]. Essa possibilidade de compartilhamento faz com que questões de segurança, como a restrição de acesso a determinados diretórios ou arquivos, sejam essenciais. As possibilidades de compartilhamento podem ser diferenciadas segundo os diferentes níveis de controle: sistema operacional (SO), sistema de arquivo distribuído (HDFS) e pela interface web (Hue).

Considerando os níveis de acesso e controle de arquivos, é importante realizar experimentos para verificar a coerência e o correto funcionamento em todos os níveis. Usualmente, testes unitários são empregados durante e após o desenvolvimento de sistemas Hadoop [Hadoop, HDFS, Hive, White 2012, Tabatabaei 2014]. Entretanto, diversos problemas referentes ao mal funcionamento e ineficiências relacionadas a segurança dos dados acabam sendo relatados por usuários durante a utilização [HadoopIssues, HDFSIssues, Tankard 2012, Bertino 2015].

Este artigo apresenta uma avaliação do controle de acesso de múltiplos usuários a múltiplos arquivos em um ambiente Hadoop envolvendo três níveis de controle e três versões da uma distribuição Hadoop. Um método de avaliação e validação é apresentado, a fim de verificar a segurança das informações considerando regras de restrição para múltiplos usuários, grupos e diversos arquivos com diferentes restrições de acesso. Experimentos são apresentados e identificamos comportamentos divergentes e errados das versões, e principalmente quando usado a interface web Hue, pela qual o comportamento foi o mesmo em todas as versões, porém, com erros.

2. Método

O método para a avaliação do controle de acesso e manipulação de arquivos foi realizado a partir de um cenário que contemplasse as diferentes formas de acesso aos arquivos. Foi criado um cenário no qual um conjunto de usuários foi organizado em grupos de modo que se pudesse avaliar o comportamento no controle de acesso de múltiplos arquivos. O objetivo dos experimentos é validar se as restrições de acesso são válidas em três níveis (SO, HDFS e Hue), considerando permissões de acesso para usuários, grupos e outros. O cenário contempla apenas permissão de leitura e escrita, já que a execução não se aplica a arquivos de dados.

A Figura 1 representa a organização de usuários e grupos no cenário proposto. São três usuários (USUARIO1, USUARIO2 e USUARIO3) e dois grupos (GRUPO1 e GRUPO2). O USUARIO1 e o USUARIO2 pertencem ao GRUPO1, e o USUARIO3 ao GRUPO2. Dessa forma, levamos em consideração as restrições que podem ser implementadas para arquivos e diretórios em relação a usuários, grupos e outros. Dessa forma é possível validar a restrição de acesso de múltiplos usuários a múltiplos arquivos.

A Figura 2 representa a forma como estão organizados os arquivos dentro do cenário, considerando a localização e tipos de restrições impostas a diretórios e arquivos. O diretório USUARIO1 permite o acesso total somente ao USUARIO1, já o

diretório GRUPO1 permite o acesso total somente aos usuários pertencentes ao GRUPO1. Um conjunto de arquivos de texto foi criado dentro de cada diretório USUARIO1 e GRUPO1, com restrições de somente leitura, somente escrita; e leitura e escrita apenas para o USUARIO1 e integrantes do GRUPO1.

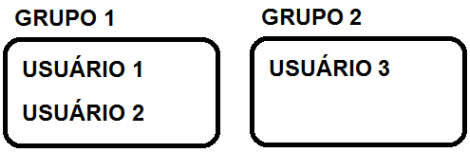


Figura 1. Distribuição de usuários por grupo.

Figura 2. Distribuição de arquivos e diretórios.

3. Experimentos e Avaliação

Os experimentos foram realizados usando três versões da distribuição Hadoop da Hortonworks - HDP (HortonWorks Data Plataform): 1.3, 2.1 e 2.2.4. O principal motivo da comparação entre as três versões é a validação do comportamento dos respectivos componentes em cada versão, as quais são:

- HDP 1.3: Hadoop 1.2.0, HDFS 1.2.0, Hue 2.2.0 e CentOS 2.6.32-358.
- HDP 2.1: Hadoop 2.4.0, HDFS 2.4.0, Hue 2.5.1 e CentOS 2.6.32-431.
- HDP 2.2.4: Hadoop 2.6.0, HDFS 2.6.0, Hue 2.6.1 e CentOS 2.6.32-504.

3.1. Execução do Método

Em cada versão do HDP foi implementado o cenário nos respectivos SO, HDFS e Hue. Ou seja, o cenário foi implementado e executado 9 vezes, objetivando eliminar possíveis desvios de comportamento, mas que não ocorreram. Os grupos, usuários e arquivos foram criados nos diferentes diretórios seguindo as restrições de acesso conforme definido pelo método (Seção 2).

Os usuários, diretórios e arquivos foram manipulados no Linux utilizando os comandos: *adduser*, *groupadd*, *usermod*, *mkdir*, *chmod*, *chgrp*, *cd* e *cat*. O HDFS utiliza os usuários e grupos do próprio SO, logo, os comandos utilizados foram os mesmos para administrar usuário no Linux. Mas, para manipular os arquivos foram utilizados os comandos específicos do HDFS, que basicamente apenas adicionam o "-" antes do comando Linux, como *–mkdir* e *–chmod*. Já no Hue, foram utilizadas as opções disponibilizadas nos menus de gerenciamento de usuários (*Hue Users*) e gerenciamento de arquivos (*File Browser)*.

3.2. Resultados

Em todas as versões do HDP (1.3, 2.1 e 2.2.4) tanto o SO quanto o HDFS asseguraram as restrições impostas aos usuários e grupos. Foram garantidas as restrições de acesso aos diretórios e arquivos, assim como as restrições de operações (leitura e escrita). Esses resultados são justificados uma vez que os usuários, grupos e restrições de acesso do HDFS são manipulados utilizando o SO.

Por outro lado, o Hue foi reprovado em diversos testes realizados (Tabela 1). Os usuários tiveram acesso a todos os diretórios e a todos os arquivos, mesmo aqueles com restrições de acesso e operações sobre os arquivos. Esses resultados se devem ao fato de que os usuários, grupos e permissões criadas pelo Hue deveriam ser implementados por uma lista de controle de acesso (ACL) no HDFS, o que não ocorreu. As primeiras versões do HDFS não implementaram ACL e mesmo a última versão testada (2.6.0) também não restringiu o acesso, mesmo criando ACLs.

Tabela 1. Resultados para o Hue.

Usuário	Diretório	Arquivo	Operação	Resposta	Resultado
USUARIO1	/USUARIO1	LEITURA.TXT	LEITURA	Permite	Aprovado
USUARIO1	/USUARIO1	ESCRITA.TXT	LEITURA	Permite	**Reprovado**
USUARIO1	/USUARIO1	LEITURA_ESCRITA.TXT	LEITURA	Permite	Aprovado
USUARIO1	/USUARIO1	LEITURA.TXT	ESCRITA	Permite	**Reprovado**
USUARIO1	/USUARIO1	ESCRITA.TXT	ESCRITA	Permite	Aprovado
USUARIO1	/USUARIO1	LEITURA_ESCRITA.TXT	ESCRITA	Permite	Aprovado
USUARIO2	/ USUARIO1	LEITURA.TXT	LEITURA	Permite	**Reprovado**
USUARIO2	/ USUARIO1	ESCRITA.TXT	LEITURA	Permite	**Reprovado**
USUARIO2	/ USUARIO1	LEITURA_ESCRITA.TXT	LEITURA	Permite	**Reprovado**
USUARIO2	/ USUARIO1	LEITURA.TXT	ESCRITA	Permite	**Reprovado**
USUARIO2	/ USUARIO1	ESCRITA.TXT	ESCRITA	Permite	**Reprovado**
USUARIO2	/ USUARIO1	LEITURA_ESCRITA.TXT	ESCRITA	Permite	**Reprovado**
USUARIO2	/GRUPO1	LEITURA.TXT	LEITURA	Permite	Aprovado
USUARIO2	/GRUPO1	ESCRITA.TXT	LEITURA	Permite	**Reprovado**
USUARIO2	/GRUPO1	LEITURA_ESCRITA.TXT	LEITURA	Permite	Aprovado
USUARIO2	/GRUPO1	LEITURA.TXT	ESCRITA	Permite	**Reprovado**
USUARIO2	/GRUPO1	ESCRITA.TXT	ESCRITA	Permite	Aprovado
USUARIO2	/GRUPO1	LEITURA_ESCRITA.TXT	ESCRITA	Permite	Aprovado
USUARIO3	/GRUPO1	LEITURA.TXT	LEITURA	Permite	**Reprovado**
USUARIO3	/GRUPO1	ESCRITA.TXT	LEITURA	Permite	**Reprovado**
USUARIO3	/GRUPO1	LEITURA_ESCRITA.TXT	LEITURA	Permite	**Reprovado**
USUARIO3	/GRUPO1	LEITURA.TXT	ESCRITA	Permite	**Reprovado**
USUARIO3	/GRUPO1	ESCRITA.TXT	ESCRITA	Permite	**Reprovado**
USUARIO3	/GRUPO1	LEITURA_ESCRITA.TXT	ESCRITA	Permite	**Reprovado**

As falhas ocorreram em todas as versões do HDP, uma vez que a interface usa um usuário administrador no HDFS e todos os usuários manipulam o sistema de arquivos por esse usuário. Assim, o controle que o HDFS possuía, sem ACL, é prejudicado, pois o usuário é único, independentemente do usuário que está registrado e manipulando os diretórios e arquivos no Hue. O controle deveria ocorrer com ACL mas não ocorreu, mesmo ativando a referida funcionalidade através da propriedade "dfs.namenode.acls.enabled" no arquivo de configuração do HDFS (hdfs-site.xml) e reiniciando o sistema.

4. Considerações Finais

Este artigo contribui com um método e um conjunto de experimentos necessários para avaliação do controle de acesso de múltiplos usuários a múltiplos arquivos em um ambiente Hadoop. Também contribui identificando comportamentos divergentes e errados das versões e principalmente quando usado a interface web Hue. Foram testadas diferentes versões da distribuição do Hadoop Hortonworks (HDP), considerando tanto o sistema operacional Linux, quanto o HDFS e a interface web Hue. Um cenário de teste foi criado e implementado considerando esses diferentes níveis de um ambiente Hadoop. Foi verificado que no Linux e HDFS o controle de acesso é realizado como esperado em todas as versões HDP. No entanto, ocorreram diferenças nas versões considerando o Hue. Nas versões HDP sem ACL, o usuário é único e isso inviabiliza o controle de acesso de diferentes usuários. Ativando ACL, as restrições de acesso a diretórios e arquivos não ocorreram. Dessa forma, são objetos futuros de estudo desse trabalho um estudo mais aprofundado do funcionamento de ACL no HDFS e o seu funcionamento com o Hue para realizar o controle de acesso a múltiplos arquivos e usuários de forma satisfatória.

Referências

Bertino, Elisa. 2015. “Big Data - Security and Privacy.” In 2015 IEEE International Congress on Big Data, IEEE, 757–61.

Hadoop. “The Apache Hadoop.” http://hadoop.apache.org/.

HadoopIssues. “Hadoop Issues Tracking.” https://issues.apache.org/jira/browse/HADOOP.

HDFS. “Hadoop Distributed File System.” http://hadoop.apache.org/hdfs/.

HDFSIssues. “HDFS Issues Tracking.” https://issues.apache.org/jira/browse/HDFS.

Hortonworks. “Hortonworks: Open Enterprise Hadoop.” http://hortonworks.com.

Hue. “Hue - Hadoop User Experience - The Apache Hadoop UI.” http://gethue.com/.

Shvachko, Konstantin, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. “The Hadoop Distributed File System.” In Proc. of the MSST - Symp. on Mass Storage Systems and Technologies, IEEE, 1–10.

Tabatabaei, Mahsa. 2014. “Evaluation of Security in Hadoop.” KTH Royal Institute of Technology.

Tankard, Colin. 2012. “Big Data Security.” Network Security 2012(7): 5–8.

Thusoo, Ashish, J.S. Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. “Hive - A Warehousing Solution Over a Map-Reduce Framework.” Proceedings of the VLDB Endowment 2(2): 1626–29.

White, Tom. 2012. Hadoop: The Definitive Guide, 3rd Edition. 3rd ed. O’Reilly Media.

Zikopoulos, Paul C., Chris Eaton, Dirk DeRoos, Thomas Deutsch, and George Lapis. 2012. Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data. McGraw-Hill.

WTF 2016
Sessão Técnica 2
Ferramentas e Sistemas para Testes e Tolerância a Falhas

Uma avaliação do potencial de detecção de defeitos dos casos de teste de robustez de acordo com a análise de mutantes

Wallace Felipe Francisco Cardoso[1] **e Eliane Martins**[1]

[1]Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)
Campinas – SP – Brazil

wallace.cardoso@ic.unicamp.br, eliane@ic.unicamp.br

Abstract. *Testing whether a system satisfies the specified functionalities is necessary, but not enough. It is also important to determine how a system behave in presence of invalid or unexpected inputs. This is the goal of robustness testing. If robustness faults go undetected during development, they can occur when the system is in operation, when faults are harder to diagnose and fix, which can lead to total interruption of system operations. In this paper we investigate the fault detection potential of nominal and robustness test sequences. Nominal sequences consider only specified inputs, whereas the robustness ones consider also innoportune inputs. We considered sequences generated from UML state diagrams which models the functional behavior of a system. Model-based mutation analysis was used to evaluate the fault detection potential of the generated sequences.*

Resumo. *Testar se o sistema satisfaz as funcionalidades especificadas é uma etapa necessária, mas não é suficiente. Importante é também determinar como o sistema se comporta em presença de entradas inválidas ou não esperadas (inoportunas). Este é o objetivo dos testes de robustez. Caso não sejam detectadas durante a fase de testes, as falhas de robustez (robustness faults) podem ocorrer durante a operação do sistema, quando são mais difíceis de diagnosticar e corrigir, e podem levar à interrupção das operações ou ao mau funcionamento do sistema. Neste trabalho analisamos o potencial de detecção de defeitos (faults) de sequências de teste nominais e sequências de teste de robustez. As sequências nominais contêm somente entradas especificadas, enquanto as sequências de robustez contêm também entradas inoportunas. As sequências são geradas a partir de modelos de estado da UML representando o comportamento funcional do sistema. Análise de mutantes foi utilizada para avaliar o potencial de detecção de falhas das sequências geradas.*

1. Introdução

Com o avanço tecnológico, os sistemas computacionais permeiam cada vez mais o nosso dia a dia. Sistemas embarcados em dispositivos diminutos são capazes de se comunicar através da internet com diversos outros sistemas. Dessa forma, nos tornamos cada vez mais dependentes destes sistemas tanto para tarefas corriqueiras, como por exemplo, trocar mensagens rápidas com outros, quanto para tarefas essenciais, como controle de tráfego aéreo, aviões e dispositivos médicos. Nestes casos, negócios ou mesmo vidas, dependem do bom funcionamento dos sistemas computacionais. Uma forma de garantir

que os sistemas se comportam conforme o que foi especificado é através da realização de testes funcionais. No entanto, determinar se o sistema realiza o que foi especificado não é mais suficiente; também interessa-nos saber se o sistema se comporta de maneira adequada em presença de falhas que podem advir das mais diversas fontes: operadores, outros sistemas, dispositivos de hardware com os quais o software interage, ou até mesmo, ataques. Este é o objetivo dos testes de robustez, que é determinar o quanto o sistema é capaz de operar adequadamente em presença de entradas inválidas ou inesperadas, ou ainda, em condições ambientais estressantes [IEEE Std 24765:2010]. Com "operar adequadamente" entende-se que o sistema deve ter mecanismos para tratar estas entradas anômalas e com isso evitar consequências catastróficas.

Os testes de robustez consistem em executar um sistema em presença de entradas anômalas que são introduzidas deliberadamente durantes os testes. Uma estratégia comumente usada nos testes de robustez consiste em introduzir parâmetros inválidos na interface do sistema com o seu ambiente [Micskei et al. 2012]. No entanto, esta estratégia não leva em conta o estado do sistema, e com isso, a controlabilidade dos testes é baixa. Nossa proposta é o uso de testes baseados em modelos tanto para gerar os testes funcionais quanto os testes de robustez.

Nos testes baseados em modelos (ou TBM), a ideia é que, a partir de um modelo formal ou semiformal do comportamento do sistema ou do seu ambiente, casos de teste completos (contendo entradas e saídas esperadas) sejam gerados [Prenninger and Pretschner 2005]. TBM é muito utilizada nos testes funcionais, e muitas propostas foram feitas, as quais utilizam o mesmo arcabouço para os testes de robustez.

O estudo apresentado neste trabalho visa responder à seguinte questão: *os casos de teste de robustez, para eventos inoportunos, detectam mais defeitos (faults) do que os casos de teste para situações normais?*

O restante deste trabalho é organizado como a seguir. A Seção 2 apresenta a fundamentação teórica. A Seção 3 apresenta alguns trabalhos relacionados, mostrando como o nosso trabalho se situa com relação ao estado da arte. A Seção 4 apresenta uma breve descrição da ferramenta StateMutest. A Seção 5 apresenta os estudos realizados visando responder à questão acima e finalmente as conclusões e perspectivas para trabalhos futuros estão contidos na Seção 6.

2. Fundamentação Téorica

Os conceitos básicos para o entendimento deste trabalho estão contidos nesta seção. Primeiro é descrita a abordagem de teste baseada em modelos, e depois uma técnica para avaliação de casos de teste, baseado no potencial de detecção de determinados defeitos típicos e específicos, conhecido como análise de mutantes.

2.1. Teste Baseado em Modelo

O objetivo da atividade de teste é descobrir a maior quantidade de defeitos presentes em um sistema. O domínio de entrada de um sistema mesmo com poucas linhas de código é enorme, o que torna a tarefa de testar sob todas as combinações possíveis de entrada algo impraticável. Por essa razão, deve-se escolher um subconjunto do domínio de entrada, a partir de informações contidas seja na própria implementação, seja

na especificação ou ainda, em um modelo do sistema. [Prenninger and Pretschner 2005, DeMillo and Offutt 1991].

Um modelo é a abstração do comportamento explícito de uma implementação chamada sistema sob teste. O modelo não deve ser tão detalhado quanto sua implementação e nem tão pobre de detalhes inviabilizando sua compreensão, mas suficiente para descrever como o sistema deverá se comportar. A etapa de criação de modelos do sistema, em uma metodologia de desenvolvimento bem planejada, antecede a etapa de criação do código do sistema. Os programadores frequentemente utilizam-se de modelos para desenvolver seu código. Assim, muitos dos erros que poderiam ocorrer futuramente podem ser evitados já durante a etapa de criação dos modelos através de uma análise, de uma simulação ou animação do modelo. Modelos também podem ser úteis na geração de casos de teste.

Existem várias propostas para a geração automática de casos de teste a partir de modelos formais (*i.e.* possuem semântica e sintaxe bem definida e formalizada). Destaca-se o uso de Máquina de Estados Finita (MEF) [Martins et al. 1999], Máquina de Estados Finita Estendida (MEFE) [Zhang et al. 2012], e Statecharts [Shirole et al. 2011, Santiago et al. 2006, Santiago et al. 2006].

Os casos de teste gerados tendo como base o modelo do sistema sob teste são abstratos (tanto quanto seu modelo). Especificamente para determinados tipos de modelos, existem alguns critérios que estabelecem requisitos mínimos de cobertura de determinados elementos, os quais podem ser combinados entre si, como por exemplo exercitar todas as transições (cobertura de transições) de uma máquina de estados como também passar por todos os estados (cobertura de estados), visando um conjunto de casos de teste potencialmente forte em detectar defeitos [Kim et al. 1999]. Os casos de teste abstratos podem ser reusados em diferentes implementações, pois independem de plataforma de execução.

Além de permitir a geração de casos de teste desde cedo no ciclo de desenvolvimento, o uso de TBM também pode fornecer uma solução para o problema do oráculo. O problema do oráculo diz respeito a como determinar a saída esperada para uma dada entrada. Tal tarefa é custosa e propensa a erros, quando realizada manualmente [Samuel et al. 2008]. O modelo, que representa o comportamento esperado do sistema, pode ser uma fonte para a produção de saídas esperadas.

Casos de teste gerados a partir de modelos são abstratos. O conjunto abstrato de casos de teste, para ser aplicado ao sistema sob teste, deve ser concretizado. É através da execução da suíte de teste concretizada que o testador conhecerá a quantidade de falhas ocorridas, e poderá tomar as medidas necessárias para a eliminação dos defeitos que as ocasionaram.

2.2. Análise de Mutantes

Análise de mutantes é uma técnica para avaliação de conjuntos de casos de teste, baseado em defeitos acidentais, introduzidos por desenvolvedores. A ideia é criar, a partir de um programa original, versões levemente alteradas sintaticamente, porém semanticamente diferentes e incorretas. Estas versões são conhecidas como *mutantes*, e são geradas através de *operadores de mutação*, os quais possuem regras do como e onde as mutações devem ser realizadas (ponto de mutação) [Jia and Harman 2011].

Pode-se observar no diagrama da Figura 1 que todos os mutantes (EQ, M1, M2,

e M3) são sintaticamente diferentes do original (OR), entretanto nem todos são semanticamente diferentes, como é o caso do mutante equivalente EQ. O mutante equivalente EQ, para qualquer entrada fornecida, sempre retornará a mesma saída do original OR. Os mutantes M1, M2 e M3 são de um único operador de mutação o qual troca um operador aritmético por outro. Neste exemplo, o mutante equivalente EQ é gerado através de um operador de mutação que troca a ordem dos parâmetros de uma função ou método.

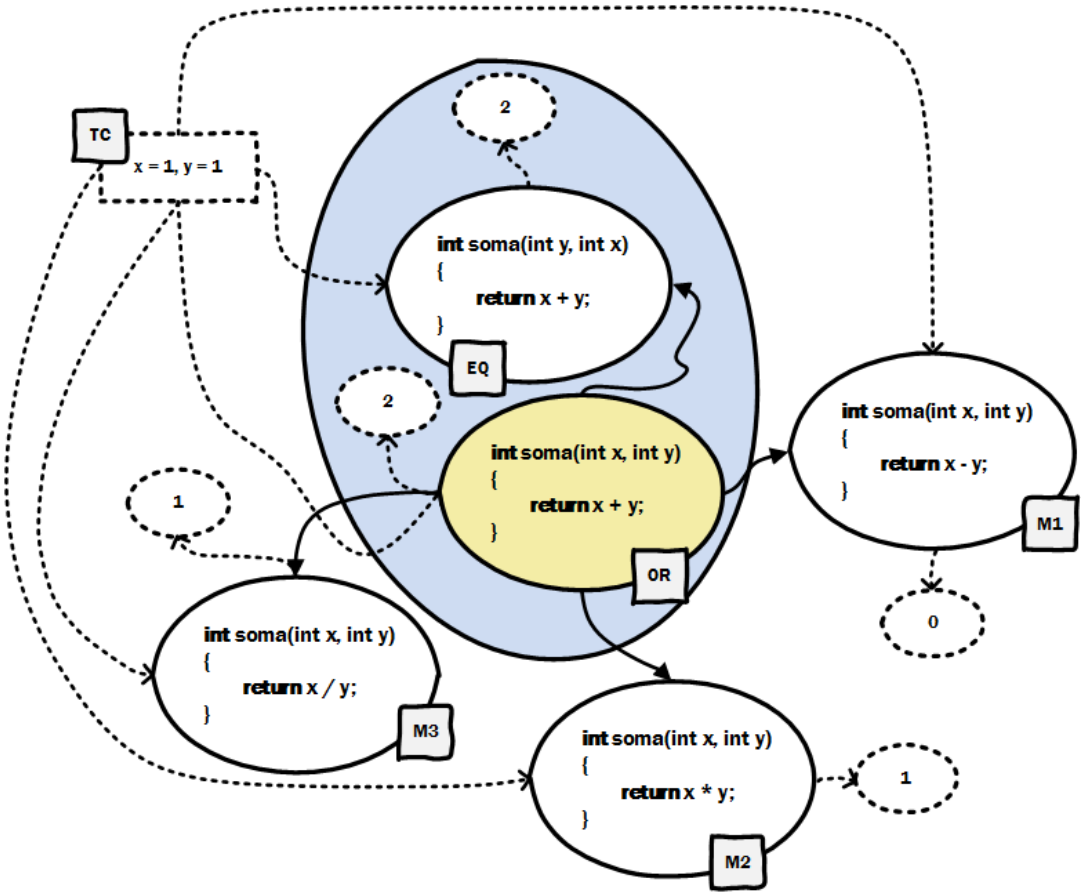


Figura 1. Programa Original (OR) e sua vizinhança (EQ, M1, M2, e M3).

Utilizando-se do conjunto de mutantes do diagrama da Figura 1, através do caso de teste TC ($x = 1, y = 1$), espera-se que a função *soma* retorne $x + y = 20$. Com TC foi possível identificar 3 possíveis versões defeituosas (M1, M2, e M3). Quando a saída do mutante difere da saída do original, o mutante é considerado *morto*. Quando a saída do mutante não difere da saída do original, para uma determinada entrada, o mutante é considerado como *vivo*. Um mutante vivo é dito *equivalente*, caso não existam entradas capazes de mostrar a diferença semântica entre ambos. Caso contrário, o conjunto de casos de teste deve ser melhorado, adicionando-se mais entradas, com o objetivo de que todos os mutantes não equivalentes sejam mortos [DeMillo and Offutt 1991]. O potencial de detecção de defeitos pode ser avaliado pelo escore de mutação, mostrado na Equação 1: onde TS é um conjunto de casos de teste, M é um conjunto de mutantes, DM é uma função que retorna a quantidade de mutantes mortos por TS, e EM é uma função que retorna a quantidade de mutantes equivalentes em M.

$$MS(TS, M) = \frac{DM(TS, M)}{|M| - EM(M)} \quad (1)$$

Claramente, o escore de mutação estabelece uma relação entre mutantes mortos e mutantes não equivalentes gerados. O ideal é ter um conjunto de teste que mate 100% dos mutantes, ou seja, com $MS = 1$.

Existem 3 condições que devem ser satisfeitas para um mutante ser morto: alcançabilidade, necessidade, e suficiência [Papadakis and Malevris 2010, Harman et al. 2011]. Alcançabilidade significa que um caso de teste deve passar pelo ponto de mutação. Necessidade significa que logo após passar pelo ponto de mutação, o mutante deve apresentar um estado interno diferente do original (infecção). Suficiência significa que a infecção interna deve se propagar até produzir uma saída observável.

Existem dois grandes problemas práticos a serem tratados ao utilizar da análise de mutantes. O primeiro é o problema da enorme quantidade de mutantes, pois existem inúmeras possibilidades de versões defeituosas mesmo para um trecho muito pequeno de código. O segundo é o problema de determinar a equivalência entre dois programas.

Duas das hipóteses propostas para a análise de mutantes restringem a geração de mutantes a apenas um subconjunto destes, conhecidas como a Hipótese do Programador Competente (HPC) e a Hipótese do Efeito Acoplamento (HEA). De acordo com a HPC, os programadores criam seus programas corretos ou muito próximo aos corretos. Como consequência da HEA, um conjunto de casos de teste que é capaz de matar mutantes formados por apenas uma única alteração sintática (mutante de primeira ordem), também é capaz de matar mutantes de várias alterações sintáticas (mutante de alta ordem) [Fabbri et al. 1994].

Mesmo utilizando-se apenas mutantes de primeira ordem, ainda assim o conjunto de mutantes é muito grande. Existem duas técnicas de redução de custos muito conhecidas que lidam com este problema: a mutação aleatória [Acree Jr 1980], e a mutação seletiva [Offutt et al. 1993]. A mutação aleatória é a seleção aleatória dos mutantes baseada em um percentual de seleção, por exemplo, selecionando-se apenas 10, 20, ou 30% dos mutantes. A mutação seletiva é a utilização criteriosa de apenas um subconjunto de operadores de mutação. Estas técnicas não são mutuamente exclusivas, e podem ser combinadas.

O problema de determinar a equivalência entre dois programas é indecidível. Existem apenas heurísticas que auxiliam o testador a determinar manualmente a equivalência entre um mutante e o original [Jia and Harman 2011].

3. Trabalhos Relacionados

Nesta seção serão apresentados alguns trabalhos que propõem ferramentas que auxiliam a atividade de teste, tanto para geração de casos de teste quanto para avaliação de casos de teste, relacionados à análise de mutantes e ao teste baseado em modelo, assim como trabalhos sobre teste de robustez e sua geração de casos de teste.

No contexto de teste de robustez e TBM, foram propostas estratégias e abordagens para geração de testes, as quais diferem na forma em que os eventos inválidos são introduzidos no modelo.

Uma estratégia consiste em completar o modelo com entradas inválidas (errôneas ou inoportunas) [Saad-Khorchef et al. 2007], bem como as saídas esperadas para estas entradas. A limitação desta estratégia é que o modelo pode se tornar muito grande para ser tratado por ferramentas de geração automática de casos de teste.

Para evitar o crescimento de modelos e com isso, a explosão combinatória do número de casos de teste, a estratégia CoFI (*Conformance and Fault Injection testing*) [Ambrosio et al. 2007] representa comportamento diante de entradas válidas e inválidas em diferentes modelos. A limitação dessa abordagem está no número de modelos, que pode crescer com a complexidade do sistema. Há ainda propostas de se modificar casos de teste, criando "mutantes" dos mesmos, para evitar a inclusão de entradas inválidas no modelo [Rollet and Salva 2009]. Neste caso, o problema é que alguns casos de teste podem se tornar infactíveis.

A proposta SABRINE (*StAte-Based Robustness testIng of operatiNg systEms*) [Cotroneo et al. 2013] extrai o modelo de estados a partir do *trace* de execução do sistema alvo, e gera os casos de teste de robustez a partir desse modelo. A limitação neste caso está no fato de que o modelo reflete o que foi implementado, e portanto, falhas de omissão de comportamento, por exemplo, não são reveladas.

Em trabalho prévio [Yano et al. 2010, Yano et al. 2011b] foi proposto um método de teste baseado em modelo de estados da UML para derivar casos de teste de forma dinâmica, *i.e.* utiliza-se um modelo executável e um algoritmo baseado em meta-heurística multiobjetivo (MOST, do inglês *Multi-Objective Search-based Testing*) o qual gera casos de teste visando cobrir um dado propósito de teste (*test purpose*), e que não sejam muito longos. Este método permite derivar casos de teste de robustez [Yano et al. 2011a] baseado na suposição de completude da UML, ou seja, por *default*, se uma entrada é recebida quando o sistema está em um estado em que essa entrada não é esperada, o sistema a ignora e permanece no mesmo estado [Rumbaugh et al. 2004].

No contexto de ferramentas de TBM, foram propostas ferramentas para a análise de mutantes e para o teste de mutação, *i.e.* avaliam ou geram conjuntos de casos de teste adequados à mutação.

A primeira ferramenta de teste para análise de mutantes baseada em modelo (*Model-Based Mutation Analysis*), de acordo com o nosso conhecimento, foi a Proteum/FSM [Fabbri et al. 1999a] . Inicialmente, os autores aplicaram a análise de mutantes manualmente em conjuntos de sequências de teste geradas a partir de MEF [Fabbri et al. 1994], propuseram novos operadores de mutação, e argumentaram como mandatório a implementação de uma ferramenta.

Em continuidade, foi proposta uma ferramenta de teste baseada em Statecharts e na análise de mutantes, chamada de Proteum/ST [Sugeta 1999] . Ambas as ferramentas Proteum/FSM e Proteum/ST implementam as técnicas de redução de custos mutação aleatória e mutação seletiva.

Momut::UML foi a primeira ferramenta em teste baseado em mutação de modelo (*Model-Based Mutation Testing*) [Aichernig et al. 2015]. Os autores propuseram uma ferramenta de geração de casos de teste, a partir dos diagramas da UML, e no contexto dos testes de mutação, no qual os diagramas utilizados são o de estados, o de instanciação, e o de classes. A saber, o conjunto de teste gerado através da Momut::UML tem o propósito

de matar mutantes, enquanto que outras ferramentas baseadas em análise de mutantes somente são capazes de avaliar os conjuntos de casos de teste de acordo com sua capacidade de matar mutantes.

Neste trabalho utilizamos a ferramenta StateMutest [Cardoso 2015], desenvolvida pelo grupo. A StateMutest oferece algumas melhorias ou diferenças significativas em relação a abordagens anteriores. O testador, através da StateMutest, não precisa completar o modelo com entradas adicionais, e também não precisa lidar com vários modelos para aspectos de robustez. Com o objetivo de reduzir a geração de casos de teste infactíveis, a ferramenta usa um método dinâmico, em que caminhos no modelo são selecionados de acordo com a execução do modelo. Outra característica da geração de casos de teste na StateMutest é a possibilidade de se obter tanto sequências nominais quanto inoportunas.

4. Sobre a ferramenta StateMutest

A StateMutest é uma ferramenta de apoio ao teste baseado em modelo [Cardoso 2015]. A StateMutest não somente oferece a geração de casos de teste a partir de modelos, mas também a avaliação do potencial de detecção de defeitos de um conjunto de teste usando a análise de mutantes. Atualmente, a StateMutest aceita apenas um subconjunto dos elementos do modelo de estados da UML. Fazem parte da ferramenta: editor gráfico e textual de MEFE, animação do modelo, assistente de importação de casos de teste (casos de teste gerados por outras ferramentas), análise de mutantes, geração de casos de teste, e possibilidade de extensões.

A geração de casos de teste de uma MEFE considera tanto o fluxo de controle quanto o de dados, o que significa considerar parâmetros, variáveis, predicados, e ações. Para tratar de um problema complexo como é o da geração de casos de teste de uma MEFE, a StateMutest utiliza-se de uma meta-heurística. A meta-heurística é multiobjetivo (MOST) [Yano et al. 2011b], a qual busca pela melhor solução baseando-se em dois objetivos. O primeiro objetivo é o de encontrar uma sequência de eventos e parâmetros que exercitem um determinado caminho do modelo, que cubra o critério dado. Por ora, somente o critério baseado em propósito de testes (*test purposes*) está disponível. O testador deve fornecer uma transição alvo (transição final a ser atingida), bem como um conjunto de cobertura contendo as demais transições a serem visitadas pelo propósito de teste. A saída são sequências de entradas que atinjam a transição alvo e que cubram uma ou mais transições do conjunto de cobertura. O segundo objetivo é o de encontrar uma solução adequada e que seja a menor possível. A razão para ter a redução do tamanho do caso de teste é por que casos de teste mais longos levam mais tempo para serem executados, dado que a ferramenta gera casos de teste de tamanhos variados, o usuário pode escolher se prefere casos de teste mais longos ou mais curtos.

A StateMutest oferece 11 operadores de mutação propostos para Statecharts [Fabbri et al. 1999b], os quais estão descritos na Tabela 1. Foram implementadas também as técnicas de redução de custos: mutação aleatória e mutação seletiva (citado na Seção 2.2). Várias tarefas simultâneas (paralelismo) são criadas durante a execução dos mutantes, já que a execução de um mutante não interfere na de outro, reduzindo assim o tempo total de execução.

Usuários experientes podem preferir criar *scripts* para automatizar os passos necessários para o uso da ferramenta. São comandos específicos da ferramenta que exe-

Tabela 1. Operadores de mutação utilizados pela StateMutest

Identificação	Nome	Descrição
TraIniStaAlt	*Transition – Initial State Alteration*	Altera o estado inicial
TraArcDel	*Transition – Arc Deletion*	Remoção de uma transição
TraEveDel	*Transition – Event Deletion*	Remoção de um evento de uma transição
TraDesStaAlt	*Transition – Destination State Alteration*	Altera o estado de destino de uma transição
OutDel	*Output Deletion*	Remove todas as ações de uma transição
StaDel	*State Deletion*	Remove um estado
CondTraConsAltCons	*Condition/Transition – Constant Alteration by Constant*	Substitui uma constante por outra constante, apenas em condições de guarda
CondTraLogOperLog	*Condition/Transition – Logical Operator by Logical Operator*	Substitui um operador lógico por outro operador lógico, apenas em condições de guarda
TraParamAltParam	*Transition – Parameter Alteration by Parameter*	Substitui um parâmetro em um transição por outro parâmetro da mesma transição
CondTraRelOperRel	*Condition/Transition – Relational Operator by Relational Operator*	Substitui um operador relacional por outro operador relacional, apenas em condições de guarda
CondTraVarAltCons	*Condition/Transition – Variable Alteration By Constant*	Substitui uma variável por uma constante, apenas em condições de guarda

cutam a geração, execução e a análise de mutantes, assim como existem os comandos específicos para geração automática de casos de teste. A criação de *scripts* é muito útil principalmente na realização de experimentos, onde várias execuções, sob as mesmas condições, são necessárias de serem realizadas sequencialmente. Usuários com conhecimento avançado de programação podem inclusive criar extensões para a ferramenta.

5. Realização dos Experimentos

Os experimentos desta seção buscam responder à questão levantada anteriormente: *os casos de teste de robustez, para eventos inoportunos, detectam mais defeitos do que os casos de teste para situações normais?* Para responder a esta pergunta, utilizamos a análise de mutantes. As sequências de entrada nominais e inoportunas são aplicadas a mutantes do modelo original e o escore de mutação é obtido. Todo o experimento foi realizado em um computador (notebook) ASUS, modelo X55C, 12 GB de memória SO-DIMM DDR3 1333Mhz, disco rígido de 7200RPM 2.5" SATA, e processador Intel Core i3-3110M 2.40Ghz. O sistema operacional utilizado foi o sistema Windows 7 Professio-

nal, e a versão da StateMutest utilizada foi a versão 3.1.30. Utilizamos o *teste-t pareado* [Walpole et al. 2011, p. 246] para as análises estatísticas, dado que o tamanho das amostras são iguais e podemos assumir que possuem a mesma variância.

Primeiramente, 11 modelos foram selecionados, os quais são considerados como *benchmark* por estarem presentes em diversas pesquisas sobre teste baseado em modelo [Kalaji et al. 2009, Yano et al. 2010]. Todos os 11 modelos (EFSM), bem como suas características, estão na Tabela 2.

Tabela 2. Os 11 modelos (EFSM) utilizados, e seus respectivos elementos

Nome do Modelo	Estados	Transições	Eventos	Guardas	Ações
ATM	10	24	13	11	51
Cashier	13	22	16	10	50
CruiseControl	6	18	10	11	110
FuelPump	14	26	17	7	112
Inres	9	19	10	5	49
Lift	7	13	12	3	12
VendingMachine	8	29	12	15	70
InFlight	5	33	15	31	313
Class2Protocol	7	22	14	11	106
ATM2	11	31	14	22	44
Inres2	6	17	8	8	78

O modelo *ATM* representa o comportamento (simplificado) de um sistema de caixa eletrônico. *Cashier* representa o comportamento de um sistema de caixa de uma loja. *CruiseControl* representa o comportamento do controlador automático de velocidade de um veículo. O modelo *FuelPump* representa o comportamento de um sistema de posto de combustível. O modelo *Inres* representa o comportamento de um protocolo de redes (simplificado). *Lift* representa o comportamento de um controlador de um elevador. *VendingMachine* representa o comportamento de um sistema de uma máquina automática de venda de produtos. *InFlight* representa o comportamento de um controlador de voo. O modelo *Class2Protocol* representa o comportamento do protocolo da camada de transporte, classe 2. Por fim, os modelos *ATM2* e *Inres2* são melhorias dos modelos *ATM* e *Inres2*, respectivamente.

Para os experimentos, foram consideradas as sequências de teste usadas em trabalhos prévios do grupo [Yano et al. 2010, Yano et al. 2011b], no qual este trabalho está inserido também. Para a obtenção dos casos de teste abstratos, aplicam-se as entradas geradas ao modelo original para obter as saídas esperadas. O conjunto de testes final foi a união dos conjuntos de testes gerados para diferentes propósitos de teste, de forma que, ao final, temos um conjunto que cobre todas as transições dos modelos utilizados. A Tabela 3 apresenta um resumo dos conjuntos de casos de teste, um para cada modelo. O menor caso de teste, de todos os modelos, tem tamanho de sequência de entrada igual a 1, estes são os casos de teste que exercitam uma das transições de saída do estado inicial. Existem também casos de teste de 4 a 5 vezes maiores do que a quantidade de transições de seus respectivos modelos, e são os que mais têm eventos inoportunos.

Após a geração dos casos de teste, utilizando a StateMutest, foi realizada a análise

Tabela 3. Resumo da quantidade de casos de teste utilizada

Nome do Modelo	Casos de Teste	Maior	Menor
ATM	70	44	1
Cashier	153	100	1
CruiseControl	94	19	1
FuelPump	71	61	1
Inres	261	103	1
Lift	52	37	1
VendingMachine	454	490	1
InFlight	326	494	1
Class2Protocol	322	482	1
ATM2	202	255	1
Inres2	188	105	1

de mutantes em dois subconjuntos de casos de teste de cada modelo. O primeiro subconjunto contém todos os casos de teste (SEI), o que significa que neste há tanto entradas nominais quanto inoportunas. O segundo é semelhante ao primeiro, exceto pela remoção de todos os eventos inoportunos (SEN). É claro que o conjunto SEI demora mais do que o conjunto SEN para executar, entretanto, a questão é qual deles é mais efetivo em detectar defeitos.

Na Tabela 4 é apresentado o resultado da análise de mutantes para o conjunto SEI e o conjunto SEN, onde NR é o conjunto de casos de teste que não alcançam o ponto de mutação do modelo (não satisfazem a propriedade de alcançabilidade), e EQ é o conjunto de mutantes equivalentes identificados manualmente. Em média, de acordo a Tabela 4, o conjunto SEI tem um potencial de detecção de defeitos maior do que o conjunto SEN ($t_{value}(11) = 6.0430 > t_{0.01} = 3.106$).

Tabela 4. Escore de mutação para o conjunto SEI e o conjunto SEN

	Escore de Mutação			
Nome do Modelo	**SEN (NR)**	**SEI (NR)**	**Casos de Teste**	**Mutantes (EQ)**
ATM	0,9626 (13)	1 (0)	70	348 (2)
Cashier	0,7197 (105)	0,8071 (74)	153	389 (3)
CruiseControl	0,8269 (9)	0,8615 (0)	94	260 (5)
FuelPump	0,9030 (17)	0,9229 (0)	71	588 (5)
Inres	0,9314 (12)	0,9838 (0)	261	248 (13)
Lift	0,9338 (10)	0,9779 (0)	52	136 (1)
VendingMachine	0,9191 (26)	0,9570 (0)	454	396 (5)
InFlight	0,7449 (136)	0,8708 (59)	326	938 (45)
Class2Protocol	0,7379 (70)	0,7964 (48)	322	393 (27)
ATM2	0,8954 (14)	0,9291 (0)	202	593 (7)
Inres2	0,8468 (9)	0,8899 (0)	188	209 (10)
Média	**0,8556**	**0,9100**		
Variância	**0,0075**	**0,0048**		

A ocorrência de eventos inoportunos não acarreta em mudança de estados. No entanto, observa-se que as sequências inoportunas alcançam melhor os pontos de mutação do que as sequências nominais.

Foi realizada a medição do tempo de execução de cada modelo considerado durante a análise de mutantes, visando descobrir se a diferença de custo entre ambos os conjuntos é significativa. O tempo de execução é apresentado na Tabela 5, onde se pode ver que o tempo de execução do conjunto SEI e do conjunto SEN, em média, não é significativamente diferente ($t_{value}(11) = 1.6831 < t_{0.05} = 2.201$).

Tabela 5. Tempo de execução dos conjuntos SEI e SEN na análise de mutantes

	Tempo	
Nome do Modelo	**SEN**	**SEI**
ATM	00:04:34	00:04:31
Cashier	00:07:43	00:08:26
CruiseControl	00:03:26	00:04:48
FuelPump	00:08:33	00:10:01
Inres	00:07:57	00:07:52
Lift	00:01:51	00:01:37
VendingMachine	00:28:05	00:27:40
InFlight	00:53:50	01:03:26
Class2Protocol	00:15:21	00:17:57
ATM2	00:18:35	00:19:37
Inres2	00:05:38	00:05:35
Média	**00:14:08**	**00:15:35**

Nosso experimento demonstra que o conjunto de casos de teste com eventos inoportunos têm um potencial de detecção de defeitos maior do que o conjunto de casos de teste que não tem eventos inoportunos, sendo o custo-benefício do primeiro melhor do que o do segundo. Embora o conjunto SEN, na maioria dos testes, execute um pouco mais rápido do que o conjunto SEI, a diferença de tempo de execução de ambos não é significativa.

6. Conclusões

Os testes de robustez, os quais testam o sistema sob condições e entradas anômalas, são uma das formas de garantir que um determinado sistema é capaz de se comportar adequadamente tanto para as condições previstas quanto para as anomalias que podem ocorrer.

Neste trabalho, através de um experimento, investigamos o potencial de detecção de defeitos dos conjuntos de casos de teste com eventos inoportunos (condições anômalas) em relação aos conjuntos de casos de teste que não os consideram (condições normais). Primeiro, geramos as sequências de entrada (nominais e inoportunas), em seguida, aplicamos as sequências geradas ao modelo, para obter as saídas esperadas para os modelos originais. Essas saídas são comparadas com as saídas produzidas quando os casos de testes gerados são aplicados aos modelos mutantes. Comparou-se então as saídas originais àquelas produzidas pelos mutantes. Depois, a análise de mutantes foi realizada visando avaliar o potencial de detecção de defeitos da suíte de teste, e responder à questão: *os*

casos de teste de robustez, para eventos inoportunos, detectam mais defeitos do que os casos de teste para situações normais?

Os resultados mostram que a capacidade de revelar defeitos dos conjuntos de eventos inoportunos foi em média superior aos testes utilizando-se apenas os eventos nominais, apresentando também um melhor custo-benefício, *i.e.* mais defeitos encontrados gastando-se pouco tempo (efetividade). Com isso, os testadores podem acrescentar sequências anômalas ou inoportunas em seus conjuntos de casos de teste, ao invés de retirá-las visando um possível ganho em desempenho, pois elas melhoram a qualidade do conjunto de testes em termos de potencial de detecção de defeitos.

Nosso experimento foi realizado com uma quantidade pequena de modelos, o que pode ser uma possível ameaça à validade. Os modelos utilizados, embora sejam parte de um *benchmark* da comunidade de teste, não são modelos reais, de sistemas reais. Os tempos mensurados também podem não ser exatos, com prejuízos, devido a possibilidade de execução de programas do sistema operacional. Portanto, não é possível generalizar os resultados desta pesquisa, contudo, conseguimos resultados motivadores em relação aos testes de robustez mesmo para um conjunto com poucos modelos disponíveis.

Futuramente, pretendemos estender a avaliação da análise de mutantes, de acordo com entradas anômalas e normais, para um conjunto que tenha uma quantidade representativa de modelos, com a presença de modelos de sistemas reais. Também pretendemos verificar se estes resultados são semelhantes utilizando-se outros critérios de teste.

7. Agradecimentos

Agradecimentos ao CNPq pelo suporte financeiro durante a realização desta pesquisa, e à Professora Doutora Sandra Camargo Pinto Ferraz Fabbri, professora do Departamento de Computação da UFSCar, por nos ajudar através de discussões sobre análise de mutantes aplicada a modelos.

Referências

Acree Jr, A. T. (1980). *On Mutation.* PhD thesis, Georgia Institute of Technology, Atlanta School of Information and Computer Science.

Aichernig, B., Brandl, H., Jobstl, E., Krenn, W., Schlick, R., and Tiran, S. (2015). Momut:: Uml model-based mutation testing for uml. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, pages 1–8. IEEE.

Ambrosio, A. M., Mattiello-Francisco, F., Santiago Jr, V. A., Silva, W. P., and Martins, E. (2007). Designing fault injection experiments using state-based model to test a space software. In *Dependable computing*, pages 170–178. Springer.

Cardoso, W. F. F. (2015). Statemutest: uma ferramenta de apoio ao teste baseado em modelos de estado estendidos. Master's thesis, Universidade Estadual de Campinas.

Cotroneo, D., Di Leo, D., Fucci, F., and Natella, R. (2013). Sabrine: State-based robustness testing of operating systems. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 125–135. IEEE.

DeMillo, R. and Offutt, A. J. (1991). Constraint-based automatic test data generation. *Software Engineering, IEEE Transactions on*, 17(9):900–910.

Fabbri, S. C., Delamaro, M. E., Maldonado, J. C., and Masiero, P. C. (1994). Mutation analysis testing for finite state machines. In *Software Reliability Engineering, 1994. Proceedings., 5th International Symposium on*, pages 220–229. IEEE.

Fabbri, S. C., Maldonado, J. C., and Delamaro, M. E. (1999a). Proteum/fsm: a tool to support finite state machine validation based on mutation testing. In *Computer Science Society, 1999. Proceedings. SCCC'99. XIX International Conference of the Chilean*, pages 96–104. IEEE.

Fabbri, S. C. P. F., Maldonado, J. C., Sugeta, T., and Masiero, P. C. (1999b). Mutation testing applied to validate specifications based on statecharts. In *Software Reliability Engineering, 1999. Proceedings. 10th International Symposium on*, pages 210–219. IEEE.

Harman, M., Jia, Y., and Langdon, W. B. (2011). Strong higher order mutation-based test data generation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 212–222. ACM.

Jia, Y. and Harman, M. (2011). An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 37(5):649–678.

Kalaji, A., Hierons, R. M., and Swift, S. (2009). Generating feasible transition paths for testing from an extended finite state machine (efsm). In *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*, pages 230–239. IEEE.

Kim, Y. G., Hong, H. S., Bae, D.-H., and Cha, S. D. (1999). Test cases generation from uml state diagrams. In *Software, IEE Proceedings-*, volume 146, pages 187–192. IET.

Martins, E., Sabião, S. B., and Ambrosio, A. M. (1999). Condata: a tool for automating specification-based test case generation for communication systems. *Software Quality Journal*, 8(4):303–320.

Micskei, Z., Madeira, H., Avritzer, A., Majzik, I., Vieira, M., and Antunes, N. (2012). Robustness testing techniques and tools. In *Resilience Assessment and Evaluation of Computing Systems*, pages 323–339. Springer.

Offutt, A. J., Rothermel, G., and Zapf, C. (1993). An experimental evaluation of selective mutation. In *Proceedings of the 15th international conference on Software Engineering*, pages 100–107. IEEE Computer Society Press.

Papadakis, M. and Malevris, N. (2010). Automatic mutation test case generation via dynamic symbolic execution. In *Software reliability engineering (ISSRE), 2010 IEEE 21st international symposium on*, pages 121–130. IEEE.

Prenninger, W. and Pretschner, A. (2005). Abstractions for model-based testing. *Electronic Notes in Theoretical Computer Science*, 116:59–71.

Rollet, A. and Salva, S. (2009). Testing robustness of communicating systems using ioco-based approach. In *Computers and Communications, 2009. ISCC 2009. IEEE Symposium on*, pages 67–72. IEEE.

Rumbaugh, J., Jacobson, I., and Booch, G. (2004). *Unified Modeling Language Reference Manual, The*. Pearson Higher Education.

Saad-Khorchef, F., Rollet, A., and Castanet, R. (2007). A framework and a tool for robustness testing of communicating software. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 1461–1466. ACM.

Samuel, P., Mall, R., and Bothra, A. K. (2008). Automatic test case generation using unified modeling language (uml) state diagrams. *Software, IET*, 2(2):79–93.

Santiago, V., Amaral, A. S. M. d., Vijaykumar, N. L., Mattiello-Francisco, M. F., Martins, E., and Lopes, O. C. (2006). A practical approach for automated test case generation using statecharts. In *Computer Software and Applications Conference, 2006. COMPSAC'06. 30th Annual International*, volume 2, pages 183–188. IEEE.

Shirole, M., Suthar, A., and Kumar, R. (2011). Generation of improved test cases from uml state diagram using genetic algorithm. In *Proceedings of the 4th India Software Engineering Conference*, pages 125–134. ACM.

Sugeta, T. (1999). *PROTEUM-RS/ST: uma ferramenta para apoiar a validação de especificações statecharts baseada na análise de mutantes.* PhD thesis, Universidade de São Paulo.

Walpole, R. E., Myers, R. H., Myers, S. L., and Ye, K. (2011). *Probability and statistics for engineers and scientists*, volume 9. Prentice Hall.

Yano, T., Martins, E., and De Sousa, F. L. (2010). Generating feasible test paths from an executable model using a multi-objective approach. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pages 236–239. IEEE.

Yano, T., Martins, E., and De Sousa, F. L. (2011a). A model-based approach for robustness test generation. In *Dependable Computing Workshops (LADCW), 2011 Fifth Latin-American Symposium on*, pages 33–34. IEEE.

Yano, T., Martins, E., and De Sousa, F. L. (2011b). Most: a multi-objective search-based testing from efsm. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 164–173. IEEE.

Zhang, J., Yang, R., Chen, Z., Zhao, Z., and Xu, B. (2012). Automated efsm-based test case generation with scatter search. In *Proceedings of the 7th International Workshop on Automation of Software Test*, pages 76–82. IEEE Press.

Arquitetura para injeção de falhas em protocolos de comunicação segura em aplicações críticas

Sergio Cechin, Taisy Silva Weber, João Cesar Netto

Instituto de Informática – UFRGS
Caixa Postal 15064 - 91501-970 Porto Alegre, RS
{taisy@inf.ufrgs.br},{cechin@inf.ufrgs.br}

Resumo: *Em aplicações industriais críticas, falhas podem provocar a morte de pessoas ou danos irreparáveis ao meio ambiente. Devido ao ambiente hostil, a comunicação é um dos elos mais frágeis do sistema. A comunicação através de protocolos TCP/IP ou barramentos de campo apresenta taxas de defeito incompatíveis com os requisitos impostos a aplicações críticas. Quando o risco de acidentes fatais é muito alto, protocolos de comunicação segura, tais como o PROFIsafe, openSafety e o Safety-over-EtherCAT, devem ser empregados. Entretanto, para cada novo equipamento, o protocolo deve ser implementado e validado obedecendo estritamente às recomendações de normas de segurança como a IEC 61508 e a IEC 61784-3. As normas exigem injeção de falhas em todas as fases de teste. Para facilitar a aplicação destas normas por parte dos desenvolvedores e testadores, o artigo propõe a arquitetura de um ambiente de injeção de falhas para validação de protocolos de comunicação sugeridos pelas normas de segurança. Para atender aos requisitos de baixo custo e alta precisão, a arquitetura proposta baseia-se no uso de hardware genérico, com o emprego eventual de adaptadores de hardware, e de software específico.*

Abstract*: In critical industrial applications, faults can cause deaths or irreparable damage to the environment. Due to the harsh environment, communication is one of the weakest components in the system. Communication via TCP/IP or fieldbus features failure rates incompatible with the requirements of the critical applications. When the risk of fatal accidents is very high, secure communication protocols, such as PROFIsafe, openSafety and the Safety-over-EtherCAT, should be employed. However, for each new device, the protocol must be implemented and validated in strict obedience to the safety standards IEC 61508 and IEC 61784-3. The standards require fault injection in all test phases. To facilitate the implementation of these standards by developers and testers, the paper proposes the architecture of a fault injection environment for validation of safety protocols. To meet the low cost and high accuracy requirements, the proposed architecture is based on the use of generic hardware, with the possible use of hardware adapters, and specific software.*

Trabalho desenvolvido no Projeto SDCD - FAURGS/UFRGS/BNDES – (13.2.0646.1 - projeto 8108).

1 Introdução

Aplicações industriais requerem o monitoramento de variáveis do processo, o processamento dessas informações segundo alguma lógica programável e a atuação sobre esse processo. Como apresentado na figura 1, o monitoramento do processo é realizado por sensores; a lógica programável é realizada por equipamentos controladores e a atuação é conduzida por dispositivos atuadores. Em sistemas mais simples, o sistema é implementado através da conexão entre sensores, controladores e atuadores através de cabos de cobre, por onde trafegam sinais elétricos proporcionais às grandezas do processo. Entretanto, nos sistemas atuais, mais complexos, a comunicação é realizada através de redes de dados.

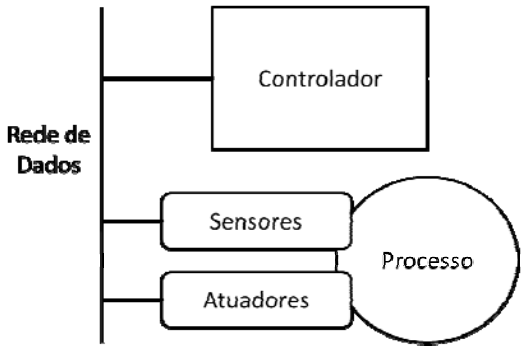


Figura 1 – Arquitetura dos Sistemas de Controle de Aplicações Industriais

Falhas de comunicação na rede de dados em aplicações *críticas*, como geração de energia, extração de petróleo, e transporte público, podem conduzir a acidentes fatais. Os controladores do processo ou controladores específicos podem ser aplicados para aumentar a segurança de instalações industriais executando funções necessárias para prevenir ou mitigar acidentes (Dunn 2003). Exemplos de funções de segurança são: controle de parada emergencial; bloqueio preventivo em maquinários pesados; sinais de alerta; e alarmes de incêndio e detecção de gases tóxicos.

Portanto, devido ao risco inerente a essas funções, elas devem receber uma atenção especial. Essa atenção aparece na forma de normas de segurança (*safety*), onde uma função de segurança é avaliada pelo seu nível integridade de segurança, SIL (Bell 2011). Integridade de segurança que é definida como a probabilidade de ocorrência de defeitos ao realizar a função de segurança, e apresenta 4 níveis. Um função SIL 4 reduz o risco de um acidente fatal em 10.000 vezes. SIL 3 reduz o risco em 1000 vezes e SIL 2 em 100 vezes. O setor de óleo e gás exige SIL 2 a SIL 3, dependendo da função. Em um cenário em que a explosão de uma plataforma de petróleo poderia provocar a morte de até 100 pessoas, se as funções de segurança forem todas SIL 3, é esperado que o dano fique restrito a menos do uma pessoa, o que é considerado tolerável.

Protocolos da pilha TCP/IP não podem ser usados para comunicação em sistemas de segurança. Eles não atendem os critérios das normas. Não há argumentação aceitável que justifique o emprego desses protocolos em ambientes críticos. Funções de segurança se executadas diretamente sobre TCP/IP impedem que os órgãos reguladores autorizem a operação do sistema. A situação não melhora se foram empregadas redes de automação como Profibus, Profinet, ou EtherCAT. A solução é o emprego de protocolos seguros de comunicação. Mas não existem protocolos seguros prontos para serem instalados em ambientes críticos, apenas especificações ou *tool kits*. Os protocolos

devem ser codificados seguindo rigorosamente as normas de segurança e tanto o processo de desenvolvimento como o código resultante devem ser certificados por uma entidade competente.

O artigo propõe uma arquitetura para um sistema de validação por injeção de falhas que permita aplicar testes sob falhas na implementação dos protocolos mais usados em ambientes industriais (como PROFIsafe, openSafety e Safety-over-EtherCAT). A arquitetura facilita a aplicação dos testes por partes dos desenvolvedores e testadores e permite reuso de cargas de falhas. A arquitetura foi recentemente implementada para validação do protocolo PROFIsafe e está sendo no momento estendida para comportar o Safety-over-EtherCAT. Este trabalho é desenvolvido com o financiamento do projeto SDCD, FAURGS/UFRGS/BNDES, e apoio da empresa Altus, que produz controladores lógicos programáveis. O SDCD visa desenvolver ambientes de controle e monitoramento distribuídos, incluindo sistemas críticos onde segurança (*safety*) é um requisito não funcional a ser atendido.

O artigo se desenvolve como segue: inicialmente são apresentadas normas de segurança que tratam de comunicação segura e é fornecida uma visão geral sobre protocolos seguros. A seguir são apresentados conceitos de injeção de falhas e alguns trabalhos relacionados e a proposta de arquitetura para o ambiente de injeção de falhas. Uma breve descrição da experiência já adquirida com o ambiente finaliza o artigo.

2 Normas de segurança

Nas últimas décadas, uma grande quantidade de normas de segurança foi proposta para vários setores de atividades (Esposito, Cotroneo, and Silva 2011). Normas diferentes são adotadas em diferentes domínios de aplicação (como óleo e gás, aviação, energia elétrica e nuclear, indústria automobilística). As normas de segurança têm em comum a obrigatoriedade do emprego de técnicas de prevenção e de tolerância a falhas para a redução de risco, além do projeto criterioso e documentado, desde as primeiras fases do ciclo de desenvolvimento do sistema. As normas permitem certificação de sistemas construídos segundo suas recomendações e que tenham sido devidamente verificados e validados.

Normas de segurança tratam tanto do hardware como do software. Software é o principal problema. Muitos acidentes graves devidos a falhas de software são reportados na literatura (Zhivich and Cunningham 2009), (Alemzadeh et al. 2013), (Hardy 2014). Aplicar criteriosamente normas de segurança não fornece garantia absoluta de evitar acidentes, mas reduz consideravelmente os riscos.

2.1 Perfis de comunicação segura em aplicações críticas

Entre as várias normas de segurança, algumas apresentam uma natureza mais genérica, como a IEC 61508 ((Fowler and Bennett 2000), (Bell 2006)). Seu principal objetivo é orientar o trabalho de equipes técnicas no desenvolvimento de sistemas computacionais de segurança, visando alcançar níveis compatíveis com os exigidos por agências reguladoras (Gall and Wen 2010). A IEC 61784-3, que padroniza perfis de comunicação segura, é derivada da IEC 61508. As normas são extensas, com centenas de medidas prescritivas baseadas em uma vasta experiência prática, pesquisas e discussões. Como consequência, tanto desenvolver um projeto em conformidade com a norma como obter certificação são tarefas árduas (Bilich and Hu 2009).

2.2 Comunicação em sistemas instrumentados de segurança

Um sistema instrumentado de segurança, SIS, é encarregado de executar a função de segurança. Geralmente, é formado por um componente lógico conectado por uma rede de comunicação a uma central e a portas remotas de entrada e saída. Essas portas remotas estão conectadas aos sensores e/ou aos atuadores, que estão por sua vez diretamente ligados ao processo. Até recentemente, SIS eram construídos com componentes discretos. A partir do ano 2000, com a publicação da IEC 61508, foi permitido que SIS fossem construídos com componentes programáveis. Atualmente, as portas remotas também são inteligentes, formando assim um sistema distribuído de instrumentação. Na figura 2 é apresentado o sistema de controle com o SIS. Nota-se a inclusão de um Controlador (*safety*), um conjunto de sensores e atuadores separados daqueles usados para o processo e uma rede segura de dados, separada da rede de dados de controle do processo.

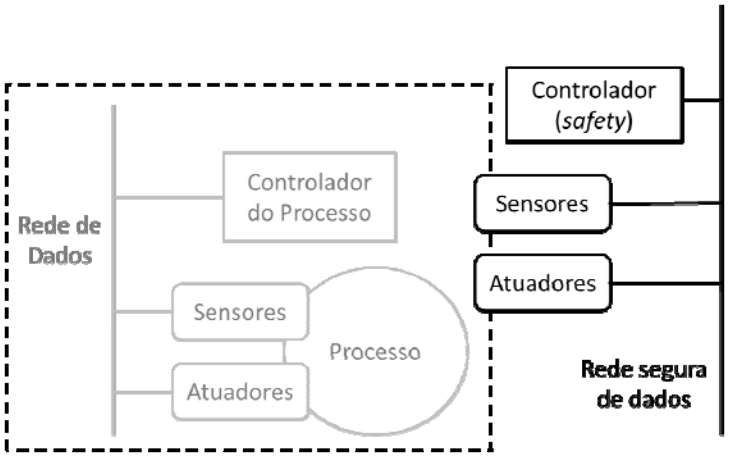


Figura 2 – SIS: sistema instrumentado de segurança

Sistemas de controle há muito tempo são programáveis, e muitos alcançam alta confiabilidade. A diferença é que apenas SIS precisam ser certificados. A separação entre sistemas de controle e de segurança reduz os custos em uma instalação industrial e, pelo isolamento de falhas, aumenta a confiabilidade. Regulamentos de vários setores, como os do setor de óleo e gás, proíbem que funções de segurança sejam executadas nos mesmos equipamentos responsáveis pelas funções de controle.

SIS programáveis são extremamente flexíveis e adaptáveis. Mas onde existe comunicação e software, existem falhas. O desafio é manter o sistema operando com segurança mesmo na ocorrência de falhas. Para evitar acidentes, deve ser garantida alta confiabilidade do próprio SIS incluindo, também, a comunicação entre seus componentes.

As normas especificam que todos os recursos de hardware, software e comunicação que implementam uma função de segurança devem ser certificados. Inicialmente, deveriam ser validadas todas as camadas de comunicação e todos os protocolos e mídias legados. Isso implicava na certificação do desenvolvimento da pilha de protocolos, desde a parte física até a aplicação, o que se mostrou inviável na prática devido a limitações de custo e de fornecedores. O problema foi resolvido na versão de 2010 da IEC 61508 com a adoção do conceito de *Black Channel*.

2.3 Parte segura e insegura da comunicação: *black channel*

É possível reduzir o risco de acidentes a valores aceitáveis dividindo a comunicação em uma parte segura e outra não segura. Na parte não segura estão os mecanismos usados para o transporte de dados. Esses operam como transportadores (*carrier*) dos pacotes de dados dos protocolos seguros e não necessitam de certificação. Esse canal é "opaco" (*black*), uma vez que, a princípio, não afeta as características de segurança da comunicação.

Os protocolos seguros são, então, implementados sobre um *black channel*, o que os torna independente dos canais físicos de transmissão, sejam eles fios de cobre, fibras ópticas, *wireless* ou *backplanes*. As taxas de transmissão e os mecanismos de detecção de erros do *black channel* não têm qualquer interferência sobre o protocolo seguro. Entretanto, o *black channel* deve garantir uma taxa máxima de defeitos. Se não operar abaixo desta taxa, não é possível garantir a segurança da camada acima.

O conceito de *black channel* é extremamente útil para o desenvolvedor. É possível, assim, usar formas de comunicação já disponíveis e se concentrar apenas na codificação da camada segura. Protocolos como EtherCAT, PROFIsafe, openSafety, INTERBUS, e FOUNDATION Fieldbus se apoiam no conceito de *black channel* (Neumann 2007).

3 Visão geral dos protocolos seguros

Os protocolos PROFIsafe, openSafety e Safety-over-EtherCAT, chamados de “perfis” de comunicação (*communication profiles*) pela IEC 61784-3, onde estão definidos, baseiam-se nos padrões das IEC 61158, IEC 61784-1 e IEC 61784-2 para barramentos de campo industriais, e da IEC 61508 para segurança funcional. Todas as especificações dos protocolos seguros mencionados neste item são pré-aprovadas para o nível de segurança funcional 3 (SIL 3).

PROFIsafe é um protocolo de comunicação criado pela PI – PROFIBUS e PROFINET Internacional. OpenSafety foi especificado e certificado pela EPSG – Ethernet POWERLINK Standardisation Group – como um protocolo aberto: quanto a especificação e documentação. Também existem algumas pilhas de protocolos disponíveis, mas que devem ser adaptadas ao hardware alvo e, então, certificadas. Por último, o Safety-over-EtherCAT foi especificado pelo ETG – EtherCAT Technology Group. É considerado um protocolo aberto. Também é conhecido pela sigla FSoE – *FailSafe-over-EtherCAT*.

Todos estes protocolos seguem o modelo de *black channel*, o que significa que são totalmente independentes do protocolo de transporte, no que diz respeito a sua capacidade em detectar e corrigir erros. Entretanto, o PROFIsafe e o Safety-over-EtherCAT estão definidos para serem transportados, a princípio, por certos protocolos específicos. No caso do PROFIsafe, a PI só garante a pré-aprovação do PROFIsafe para segurança funcional, caso seja usado o PROFIBUS ou o PROFINET como protocolo de transporte. No caso do Safety-over-EtherCAT, as características de tempo real da comunicação só são garantidas quando se usa uma infraestrutura EtherCAT para o transporte. Por outro lado, o openSafety não tem essas restrições, podendo operar sobre vários protocolos de transporte, incluindo PROFINET e Ethernet/IP.

3.1 Modelo de falhas e mecanismos de detecção dos protocolos seguros

Segundo as normas, os protocolos seguros devem prover mecanismos para detectar e corrigir erros provocados por falhas nas mensagens transmitidas: corrupção, repetição, perda, inserção, troca de ordem, atraso e mascaramento (interpretação de mensagens seguras como sendo não-seguras e vice-e-versa). O modelo de falhas é semelhante ao modelo adotado por protocolos convencionais de comunicação e se concentra sobre os problemas que afetam as mensagens que transitam na rede de dados. A detecção de erros também é semelhante, mas a cobertura dos mecanismos de detecção deve ser maior, porque é a cobertura que vai definir a segurança do sistema.

Para fazer frente à tarefa de detecção, o PROFIsafe utiliza o conceito de conexão e, sobre as mensagens trocadas nessa conexão, são aplicados mecanismos para verificar a consistência das mensagens, um sistema de nomes único para emissor e receptor, temporização das mensagens e um mecanismo virtual para numeração das mensagens seguras.

O Safety-over-EtherCAT também trafega suas mensagens seguras sobre uma conexão, em que são empregados mecanismos de numeração sequencial, para identificação de perdas, inserções ou repetições e CRC, para detectar a eventual corrupção das mensagens. Ainda, de forma semelhante ao PROFIsafe, as conexões e sessões de operação são unicamente identificadas. Finalmente, de forma diferente do PROFIsafe, o Safety-over-EtherCAT utiliza um relógio global e rótulos de tempo nas mensagens. No caso do openSafety, à semelhança do Safety-over-EtherCAT, são usados rótulos de tempo nas mensagens, identificadores únicos para as mensagens e a integridade dos dados das mensagens é verificada através de CRC.

3.2 Implementação de protocolos de comunicação seguros

O protocolo seguro (norma IEC 61784-3) deve ser implementado seguindo as cláusulas de desenvolvimento de software da IEC 61508. Alguns autores (Mayr, Plosch, and Saft 2011) afirmam não existir evidências suficientes de que a aplicação das normas conduza a software mais seguro. Lloyd e Reeve (Lloyd and Reeve 2009) listam várias dificuldades que as empresas enfrentam para se adaptar a IEC 61508 e obter a certificação do primeiro produto. As dificuldades passam pela falta de qualificação e competência da equipe de desenvolvimento e validação, pouca familiaridade com tolerância a falhas e engenharia de software, falhas na documentação, práticas usuais de desenvolvimento muito diferentes das preconizadas pela norma, impossibilidade de rastrear requisitos, uso de código legado, e adoção de ciclo de vida inadequado. Uma vez, entretanto, que consigam a primeira certificação, os demais produtos são mais facilmente certificados devido à mudança na cultura de desenvolvimento introduzida na empresa.

Para conseguir certificação, entretanto é preciso seguir as normas. Uma questão é a necessidade de desenvolvimento dos protocolos "a partir do zero". Existem fortes restrições de codificação (Vidal et al. 2014). Um determinado código só pode ser certificado se todo o seu processo de desenvolvimento foi realizado segundo as técnicas recomendadas na norma e se foi verificado pelos órgãos certificadores. Para evitar uma implementação completa, existem pilhas de protocolos de segurança pré-aprovadas ("aprovado" é diferente de "certificado"). Estas, por sua vez, devem ser ajustadas ao protocolo de transporte e ao hardware que lhe dará suporte. Essa forma de operação tem consequências: (1) alguns protocolos de segurança requerem que sejam utilizados

protocolos específicos para transporte. É o caso do EtherCAT e o PROFIsafe; e (2) custo de desenvolvimento, mesmo com o uso de pilhas aprovadas não é tão fácil e rápido quando se poderia esperar, pois é necessário desenvolver (e certificar) o processo de adaptação dessas pilhas ao hardware de suporte.

As normas obrigam obedecer requisitos para implementação e validação seguindo um ciclo de vida que enfatiza verificação e validação da implementação do protocolo. Entre as estratégias de validação determinadas pela norma está a injeção de falhas. Finalmente, depois da implementação, validação e verificação, o código executando em um dado equipamento poderá ser certificado. Note-se que só é possível certificar uma implementação existente. Ou seja, uma pilha genérica de protocolos não pode ser certificada. No máximo, ela será "aprovada" para uso no desenvolvimento que leva a uma implementação, que então poderá ser certificada.

3.3 Ciclo de vida de desenvolvimento

A IEC 61508 apresenta um ciclo de vida global para o sistema de segurança e ciclos de vida para a realização do software e do hardware do SIS. A implementação do protocolo de segurança deve seguir o ciclo de vida de desenvolvimento de software. Os ciclos de hardware e software são semelhantes e envolvem as fases de especificação dos requisitos de segurança, planejamento e validação de segurança, projeto e desenvolvimento, integração hardware e software, procedimentos para operação e modificação e finalmente validação de segurança. Todos os ciclos e fases são extremamente convencionais. Técnicas inovadoras, como métodos ágeis, projeto orientado a objetos, aprendizagem de máquina, reconfiguração dinâmica, não são bem aceitas na área de desenvolvimento de sistemas seguros.

A fase de desenvolvimento de software se desdobra no *diagrama V* (figura 3) com um maior detalhamento de subfases.

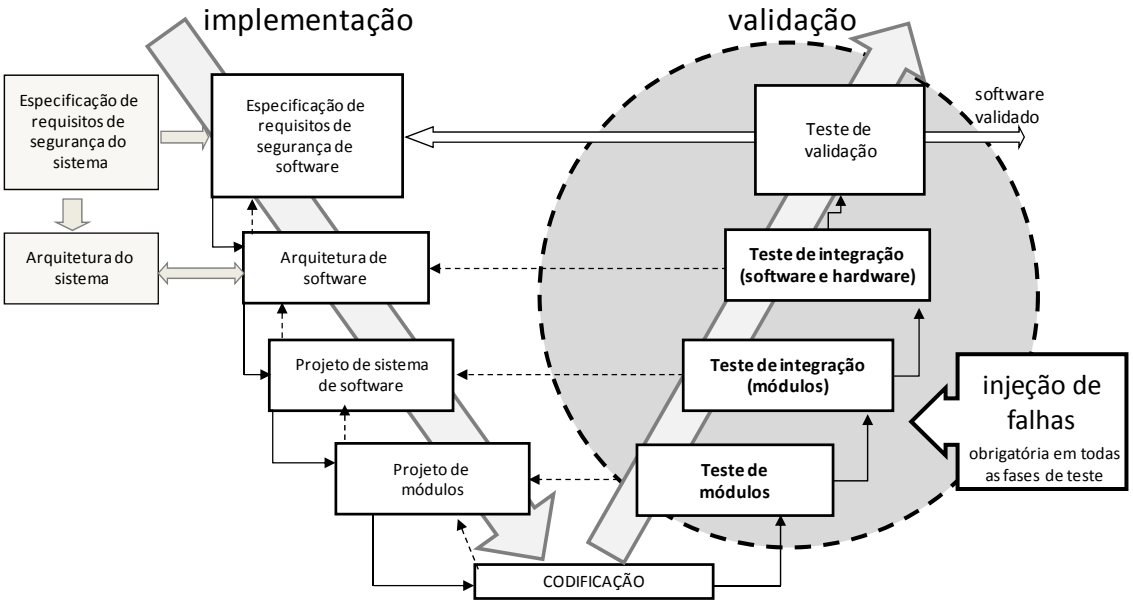


Figura 3 – Diagrama V para desenvolvimento de software na IEC 61508

Fases importantes do ciclo de vida de desenvolvimento do protocolo seguro são relacionadas a tarefas de validação, que corresponde ao lado direito do diagrama V

(figura 3). Todas as formas tradicionais de teste são aplicadas para validação, incluindo teste unitário, teste de integração dos módulos e teste de integração no equipamento alvo (hardware e software). Todos os requisitos devem ser rastreáveis desde a especificação até o código relacionado a cada requisito e ao teste do requisito, assim como no sentido inverso.

Uma técnica de teste imposta pelas normas para as fases de validação é a injeção de falhas. As falhas previstas no modelo de falhas de comunicação devem ser emuladas e o comportamento do protocolo seguro sob falhas deve ser avaliado. Desta forma é possível medir se a cobertura de falhas dos mecanismos de detecção e recuperação de erros está em conformidade com a especificada para um dado SIL.

A injeção de falhas não substitui as demais estratégias de teste e verificação, mas as complementa. Ela é essencial para determinar se os requisitos de segurança especificados foram atingidos e se a tolerância a falhas implementada aumenta a segurança, confiabilidade e disponibilidade do sistema (Pintard et al. 2013).

4 Injeção de falhas e trabalhos relacionados

As estratégias de tolerância a falhas do protocolo de comunicação devem ser ativadas para avaliar sua correção. Como a detecção de falhas só entra em operação na ocorrência de falhas, tais falhas devem estar presentes durante a avaliação do protocolo. Para não depender da ocorrência natural de falhas no ambiente real de operação, uma técnica eficaz (Hsueh, Tsai, and Iyer 1997) é emular falhas por software e injetá-las no protocolo alvo do teste. Através da introdução controlada de falhas, é possível determinar se o protocolo tolera ou não as falhas injetadas.

Um experimento de injeção de falhas é um teste executado em condições controladas na qual são introduzidas falhas. Seguindo uma carga de falhas pré-definida, falhas são introduzidas no sistema durante a execução de uma carga de trabalho. Ambas a carga de trabalho e a carga de falhas devem ser representativas do uso do sistema em condições reais de operação. Para avaliação de protocolos de comunicação, a carga de trabalho são as mensagens trocadas entre os componentes da rede e a carga de falhas inclui as falhas que afetam a troca de mensagens e estão previstas na especificação do protocolo, como por exemplo: corrupção, repetição, perda, inserção, troca de ordem, atraso e mascaramento de mensagens.

Em um experimento, parâmetros que podem ser variados para a carga de trabalho incluem número, tamanho dos pacotes e frequência de transmissão. Parâmetros de carga de falhas incluem: tipos de falhas, instantes de ativação, frequência de ocorrência de falhas e distribuição de probabilidade. Um ambiente experimental de injeção de falhas injeta falhas na troca de mensagens e monitora o comportamento do protocolo enquanto executa uma carga de trabalho.

Um problema neste tipo de teste é a falta de injetores adequados. Apesar dos vários injetores reportados (Natella, Cotroneo, and Madeira 2016), ferramentas funcionais e com usabilidade adequada são difíceis de serem encontradas. A maioria das ferramentas apresenta problemas de portabilidade (Schirmeier et al. 2012) que vão desde terem sido desenvolvidas para ambientes muito específicos, a exigirem sistema operacional ou bibliotecas específicas. Grande parte dos injetores não tem código disponível para uso ou não é mais mantido por seus desenvolvedores.

Na literatura, são encontrados alguns trabalhos de avaliação de tolerância a falhas em protocolos de comunicação ((Dawson et al. 1996), (Hurtig and Brunstrom 2008), (Siqueira et al. 2009)). Também são encontrados trabalhos que discutem a importância da injeção de falhas no desenvolvimento e certificação de sistemas críticos, ((Pintard et al. 2013), (Cotroneo and Natella 2013)). Os trabalhos mais próximos do relatado aqui são injetores de falhas para redes de controle. *Fault Injection Framework for PROFIBUS*, uma rede de campo que suporta PROFIsafe, é um framework de injeção de falhas desenvolvido para avaliar o PROFIBUS (Carvalho, Carvalho, and Portugal 2005). Seu principal objetivo é analisar o comportamento do PROFIBUS na ocorrência de falhas e também a interferência dos mecanismos de tolerância a falhas no desempenho do protocolo. O sfiCAN - *star-based physical fault injector for CAN* (Gessner et al. 2014) é um injetor de falhas físicas para o protocolo CAN, que tem por objetivo testar o comportamento dos nodos de uma rede CAN na presença de erros de canal, em particular dos controladores de nodos CAN e do software executando neles. Apesar da proximidade, nenhum dos trabalhos citados contempla os requisitos necessários para o nosso ambiente de avaliação.

5 Arquitetura do ambiente de avaliação

A arquitetura deve permitir a validação por injeção de falhas de qualquer protocolo seguro construído sobre um *black channel*, definido conforme a IEC 61784-3. Portanto, com modelos de falhas idênticos, a mesma carga de falhas dos experimentos de teste pode ser reutilizada na validação desses vários protocolos. Entretanto, os protocolos seguros executam sobre infraestruturas diferentes. Não existe hardware comum nem software comum de apoio ao protocolo. A solução adotada em injetores de uso geral, como é o caso do Firmament (Drebes et al. 2006), que operam interceptando mensagens no kernel do sistema operacional da máquina que executa o protocolo, não é adequada para protocolos seguros.

Assim, para sistematizar o estudo dos aspectos relacionados ao emprego dos injetores de falhas em sistemas críticos, os ambientes de avaliação foram classificados segundo dois aspectos:

- quanto ao tipo de interferência causada pelo injetor: pode ser classificada como “interferência espacial” ou “interferência temporal”;
- quando a localização de aplicação do injetor: pode ser de aplicação interna aos dispositivos ou externa aos dispositivos.

5.1 Sobre o tipo de interferência do injetor

As máquinas onde são implementados o transmissor e o receptor das mensagens dos protocolos seguros podem ter diferentes tipos de *kernel* de sistema operacional ou mesmo não ter sequer sistema operacional. Além disso, mesmo se houvesse um *kernel* ou outro módulo de software comum a todos os protocolos, o injetor de falhas não poderia estar embutido nesse software. O motivo disso é que todo o software embarcado que estiver associado à função de segurança tem que ser certificado. Portanto, se o injetor de falhas estiver embutido na aplicação, ele deveria ser certificado, o que não faz sentido, pois não é necessário para a implementação da função de segurança, nem é economicamente viável, pois a certificação tem custos que crescem com a complexidade do software empregado. Esse tipo de interferência onde o injetor

representa a ocupação de memória é denominado de *interferência espacial*, que não se pode aceitar no caso dos sistemas críticos.

O segundo aspecto que deve ser considerado na definição do ambiente de injeção de falhas é a *interferência temporal*: interferência do injetor no tempo de propagação das mensagens trocadas pela rede, o que pode levar a erros de temporização do protocolo sob teste. Notar que essa interferência é típica dos protocolos de tempo real, que é o caso dos protocolos seguros, devido a sua grande aplicação em ambientes industriais. A interferência temporal do injetor deve ser compensada para que o protocolo não acuse falhas no seu comportamento funcional durante os experimentos de injeção de falhas.

5.2 Sobre a localização de aplicação do injetor

Um injetor pode ser desenvolvido para *aplicação interna* aos dispositivos do sistema. Esse enfoque oferece a vantagem do baixo custo de implementação, pois não requer o acréscimo de hardware específico. Entretanto, essa técnica sempre causa interferência espacial, que deve ser mantida em níveis aceitáveis. Por outro lado, pode-se reduzir significativamente a interferência temporal, buscando formas de compensar eventuais atrasos ou avanços introduzidos pelo injetor no tratamento das mensagens do protocolo.

Como opção para a aplicação interna, pode-se recorrer à aplicação *externa do injetor*. Nesse caso, a desvantagem está na necessidade de um hardware específico, onde deve rodar uma pilha de protocolos capaz de decodificar as mensagens, injetar a falha e recodifica-las. A vantagem desse enfoque é a inexistência de interferência espacial e a possibilidade de se controlar a interferência temporal usando as técnicas semelhantes àquelas empregadas em um injetor de aplicação interna.

Em linhas gerais, a aplicação interna dos injetores oferece um baixo custo de implementação, quando comparado com a aplicação externa. Entretanto, a aplicação interna dos injetores apresenta níveis de interferência maiores do que a aplicação externa.

Com relação aos protocolos de segurança empregados nos sistemas críticos, é importante ressaltar a questão do custo de certificação: a aplicação interna pode requerer que o injetor seja certificado, o que não faz sentido, conforme já explicado. Então, a localização de aplicação do injetor deve ser tal que não seja necessária essa certificação. Isso só é possível se o injetor de falhas de comunicação estiver aplicado no *black channel*. Dessa forma, pode-se ter injetores de aplicação externa, pois os canais de comunicação sempre estão no *black channel*, ou injetores de aplicação interna, desde que na parte interna do *black channel*.

Mesmo com a possibilidade de aplicar os injetores internamente aos dispositivos, no *black channel*, essa opção apresenta dificuldades de aplicação. A injeção de falhas pode ser usada, entre outras, para verificação de dispositivos para os quais não se tem acesso ao seu interior. Dessa forma, não é possível aplicar o injetor à parte do *black channel* existente no dispositivo. Assim, a arquitetura do ambiente requer um injetor externo aos dispositivos, tornando-o agnóstico às características do dispositivo sob teste (fabricante, modelo, etc).

5.3 Arquitetura dos injetores externos

Os injetores externos são implementados através de equipamentos específicos e aplicados externamente aos dispositivos sob teste. Portanto, não apresentam interferência espacial. Esses injetores externos podem ser construídos de várias formas, no que diz respeito às características de seus componentes de hardware e software. Cada forma de implementação oferece diferentes níveis de interferência temporal.

Injetores com hardware específico usam software apenas para a sua configuração. Toda a atuação é realizada pelo próprio hardware. São os injetores que oferecem a menor interferência temporal. Entretanto, em geral, são os menos flexíveis, permitindo a aplicação de poucos tipos de falhas. Dessa forma, oferecem muitas dificuldades para injetar uma carga de falhas como aquela necessária para o teste de protocolos de segurança. Esses são injetores que não utilizam software no processo de injeção de falhas.

Quando se acrescenta software na implementação do injetor, encontra-se a categoria dos injetores com hardware e software específicos. No caso dos injetores de falhas em protocolos de comunicação, esses equipamentos são construídos de maneira que o hardware e o software se conjugam para aplicar cargas de falhas com a maior precisão possível. Apesar desse ótimo desenho, são também os mais caros.

No outro extremo encontram-se os injetores construídos a partir de hardware e software genéricos. Esses injetores permitem ao analista a construção de sua própria carga de falhas, a um custo significativamente menor do que aqueles com hardware específico: são extremamente flexíveis. Entretanto, dependendo do protocolo a ser testado, é necessário acrescentar pequenos adaptadores de hardware. Outra desvantagem desses injetores é que se tem pouco controle da interferência temporal, o que faz dele um injetor aceitável para teste de protocolos comuns de comunicação, mas pouco adequados para a injeção de falhas em protocolos de segurança.

5.4 Ambiente de avaliação

Para acomodar os requisitos de baixo custo e alta precisão, a arquitetura proposta para o ambiente baseia-se no uso de hardware genérico, com o uso eventual de adaptadores de hardware, e software específico. Com essa arquitetura pode-se construir injetores de custo comparável àquele dos injetores de hardware e software genéricos. Apesar disso, os injetores oferecem precisão comparável àquela dos injetores de hardware e software específicos, inclusive com a um bom controle sobre a interferência temporal.

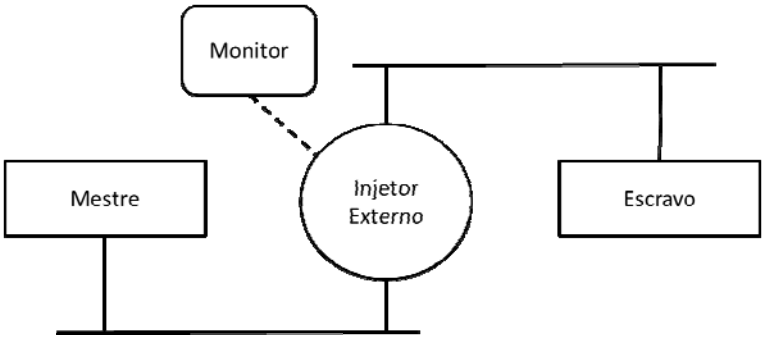


Figura 4 – Aplicação dos injetores externos

Na figura 4 está esquematizado o ambiente de aplicação proposto. No ambiente, aparecem um nodo "Mestre", responsável por iniciar qualquer comunicação, e um "Escravo", que sempre deve responder às solicitações do mestre. Esse modelo de comunicação visa garantir um alto nível de segurança das informações. O injetor de falhas externo aparece interceptando as mensagens trocadas entre mestre e escravo, de tal forma que nenhum deles é capaz de perceber a existência do injetor. Finalmente, os procedimentos de injeção de falhas podem ser configurados, controlados e monitorados, de maneira a promover a realização dos testes especificados.

Essa arquitetura foi utilizada em um trabalho de mestrado (Dobler 2016) do grupo de pesquisas, na implementação de um injetor de falhas para o protocolo "PROFIsafe". Esse injetor será utilizado nas etapas de validação das implementações do protocolo e na verificação da compatibilidade entre equipamentos de diferentes fabricantes.

6 Conclusão

As normas de segurança estabelecem os requisitos necessários para assegurar que os sistemas sejam projetados e operados para suprir as exigências do SIL requerido para uma dada função de segurança. A comunicação é um componente importante de um sistema de instrumentação distribuído e a implementação dos protocolos de segurança precisam seguir as normas.

No Brasil, a crescente demanda por equipamentos certificados faz com o desenvolvimento de software seguro seja igualmente impulsionado. No momento em que as normas de segurança funcional permitiram o uso de controladores programáveis, uma vasta gama de oportunidades se abriu para desenvolvedores de software. Entretanto a norma restringe a aplicação das técnicas e procedimentos limitando-as àquelas comumente usadas por desenvolvedores e testadores de aplicações convencionais, o que torna a implementação e validação de protocolos seguros uma tarefa árdua.

As normas de segurança exigem o emprego de injeção de falhas. Para facilitar essa tarefa, no âmbito do projeto SDCD-NG, está em fase final de desenvolvimento um injetor de falhas para validação da implementação de protocolos seguros, que segue o ambiente de validação proposto neste artigo. Nesta primeira versão do ambiente escolheu-se validar implementações do PROFIsafe seguindo demandas da empresa parceira do projeto. A próxima extensão prevista no projeto para o ambiente deverá permitir validação do Safety-over-EtherCAT.

Para atender as necessidades do projeto, a arquitetura atende aos requisitos de baixo custo e alta precisão. Estes requisitos foram alcançados através de uma criteriosa fusão entre as características de injetores com hardware e software específicos, que são muito precisos porem caros, com injetores com hardware e software genéricos, que oferecem os menores custos. A arquitetura proposta neste artigo foi empregada no desenvolvimento de um injetor real para o protocolo PROFIsafe e está atualmente em fase final de depuração.

7 Bibliografia

Alemzadeh, Homa, Jai Raman, Zbigniew Kalbarczyk, and Ravishankar Iyer. 2013. "Analysis of Safety-Critical Computer Failures in Medical Devices." *IEEE Security & Privacy* 99 (1): 1.

Bell, Ron. 2006. “Introduction to IEC 61508.” In *Proceedings of the 10th Australian Workshop on Safety Critical Systems and software-Volume 55*, 3–12. Australian Computer Society, Inc.

———. 2011. “Introduction and Revision of IEC 61508.” In *Advances in Systems Safety*, edited by Chris Dale and Tom Anderson, 273–91. Springer London.

Bilich, Carlos, and Zaijun Hu. 2009. “Experiences with the Certification of a Generic Functional Safety Management Structure According to IEC 61508.” In *Computer Safety, Reliability, and Security*, edited by Bettina Buth, Gerd Rabe, and Till Seyfarth, 5775:103–17. Lecture Notes in Computer Science. Springer Berlin / Heidelberg.

Carvalho, José, Adriano Carvalho, and Paulo Portugal. 2005. “Assessment of PROFIBUS Networks Using a Fault Injection Framework.” *Proceedings of 10th IEEE Conference on Emerging Technologies and Factory Automation, ETFA 2005*, 415–23.

Cotroneo, Domenico, and Roberto Natella. 2013. “Fault Injection for Software Certification.” *Security & Privacy, IEEE* 11 (4): 38–45.

Dawson, S., F. Jahanian, T. Mitton, and T. L. Tung. 1996. “Testing of Fault-tolerant and Real-time Distributed Systems via Protocol Fault Injection.” In *Fault Tolerant Computing, 1996., Proceedings of Annual Symposium On*, 404–14.

Dobler, R.J. 2016. "FITT: Uma Ferramenta de Injeção de Falhas para Validar Protocolos de Comunicação Seguros". Dissertação de mestrado. UFRGS.

Drebes, R.J., Gabriela Jacques-Silva, Joana Matos Fonseca Da Trindade, and Taisy Silva Weber. 2006. “A Kernel-based Communication Fault Injector for Dependability Testing of Distributed Systems.” In *1st International Haifa Verification Conference on Hardware and Software, Verification and Testing, November 13, 2005 - November 16, 2005*, 3875 LNCS:177–90. Lecture Notes in Computer Science. Haifa, Israel: Springer Verlag.

Dunn, W.R. 2003. “Designing Safety-critical Computer Systems.” *Computer* 36 (11): 40– 46.

Esposito, Christian, Domenico Cotroneo, and Nuno Silva. 2011. “Investigation on Safety-Related Standards for Critical Systems.” In *Software Certification (WoSoCER), 2011 First International Workshop On*, 49–54. IEEE.

Fowler, Derek, and Phil Bennett. 2000. “IEC 61508 — A Suitable Basis for the Certification of Safety-Critical Transport-Infrastructure Systems ??” In *Computer Safety, Reliability and Security*, edited by Floor Koornneef and Meine van der Meulen, 1943:250–63. Lecture Notes in Computer Science. Springer Berlin / Heidelberg.

Gall, Heinz, and Joachim Wen. 2010. “Functional Safety IEC 61508 and Sector Standards for Machinery and Process Industry the Impact to Certification and Users Including IEC 61508 2nd Edition.” In *23rd International Congress on Condition Monitoring and Diagnostic Engineering Management, COMADEM 2010, June 28, 2010 - July 2, 2010*, 73–81. Nara, Japan: Sunrise Publishing Limited.

Gessner, D., M. Barranco, A. Ballesteros, and J. Proenza. 2014. “sfiCAN: A Star-Based Physical Fault-Injection Infrastructure for CAN Networks.” *IEEE Transactions on Vehicular Technology* 63 (3): 1335–49.

Hardy, Terry L. 2014. “Case Studies in Process Safety: Lessons Learned from Software related Accidents.” *Process Safety Progress* 33 (2): 124–30.

Hsueh, Mei-Chen, T.K. Tsai, and R.K. Iyer. 1997. “Fault Injection Techniques and Tools.” *Computer* 30 (4): 75–82.

Hurtig, Per, and Anna Brunstrom. 2008. “Enhancing SCTP Loss Recovery: An Experimental Evaluation of Early Retransmit.” *Computer Communications* 31 (16): 3778–88.

Lloyd, M. H., and P. J. Reeve. 2009. “IEC 61508 and IEC 61511 Assessments-some Lessons Learned.” *4th IET International Conference on Systems Safety*, 2A1.

Mayr, Alois, Reinhold Plosch, and Matthias Saft. 2011. “Towards an Operational Safety Standard for Software: Modelling IEC 61508 Part 3.” In *18th IEEE International Conference and Workshops on Engineering of Computer-Based Systems, ECBS 2011, April 27, 2011 - April 29, 2011*, 97–104. Las Vegas, NV, United states: IEEE Computer Society.

Natella, Roberto, Domenico Cotroneo, and Henrique S. Madeira. 2016. “Assessing Dependability with Software Fault Injection: A Survey.” *ACM Computing Surveys (CSUR)* 48 (3): 44.

Neumann, Peter. 2007. “Communication in Industrial automation—What Is Going On?” *Control Engineering Practice* 15 (11): 1332–47.

Pintard, Ludovic, Jean-Charles Fabre, Karama Kanoun, Michel Leeman, and Matthieu Roy. 2013. “Fault Injection in the Automotive Standard ISO 26262: An Initial Approach.” In *Dependable Computing*, edited by Marco Vieira and João Carlos Cunha, 126–33. Lecture Notes in Computer Science 7869. Springer Berlin Heidelberg.

Schirmeier, H., M. Hoffmann, R. Kapitza, D. Lohmann, and O. Spinczyk. 2012. “Fail #x2217;: Towards a Versatile Fault-injection Experiment Framework.” In *ARCS Workshops (ARCS), 2012*, 1–5.

Siqueira, Torgan, Bruno Fiss, Raul Weber, Sergio Cechin, and Taisy Weber. 2009. “Applying FIRMAMENT to Test the SCTP Communication Protocol Under Network Faults.” In *2009 10th Latin American Test Workshop, LATW 2009, March 2, 2009 - March 5, 2009*. Rio de Janeiro, Brazil: Inst. of Elec. and Elec. Eng. Computer Society.

Vidal, William, Rodrigo Dobler, Sérgio Cechin, Taisy Weber, and João Netto. 2014. “Aplicação Da IEC 61508 Na Prototipação de Protocolos Seguros de Comunicação.” In *Workshop de Testes e Tolerância a Falhas*, 147–59. Florianópolis: Sociedade Brasileira de Computação.

Zhivich, Michael, and Robert K. Cunningham. 2009. “The Real Cost of Software Errors.” *Security & Privacy, IEEE* 7 (2): 87–90.

DETOX: Detecção de Inconsistências na Política de Segurança Implementada em Firewall Real

Ygor Kiefer Follador de Jesus Magnos Martinello Eduardo Zambon

[1]NERDS - Núcleo de Estudos em Redes Definidas por Software
UFES - Universidade Federal do Espírito Santo, Campus Goiabeiras

ygor.jesus@ufes.br, magnos@inf.ufes.br, zambon@inf.ufes.br

Abstract. *It is a complex and demanding task to ensure the consistency of rules that implement a network security policy using a firewall. In particular, when such task is not performed properly, network vulnerabilities can arise. In this paper, we present the DETOX tool, a software that detects inconsistencies among rules that constitute a firewall. First, we validate the tool implementation by reproducing and extending the results presented in the literature. After such validation, we perform a real life case study, by analysing the firewall configuration currently in use at a university. During this investigation, the tool discovered several inconsistencies previously unknown.*

Resumo. *Garantir a consistência das regras que implementam uma política de segurança de rede através de um* firewall *é uma tarefa complexa e exaustiva, podendo gerar vulnerabilidades na rede quando mal executada. Neste artigo, realizamos um estudo de teorias e algoritmos já existentes nesta área e apresentamos o DETOX, uma ferramenta para a detecção de inconsistências entre regras que compõem um* firewall*. Além de validar sua implementação reproduzindo os resultados obtidos na literatura, a ferramenta foi utilizada em um caso de estudo real, aplicado sobre a configuração de* firewall *em uso em uma instituição de ensino superior. Neste caso de estudo, a ferramenta indicou a existência de inconsistências que não haviam sido previamente descobertas.*

1. Introdução

Com o aumento do número de pessoas com acesso à Internet, a quantidade de informações trafegando pela rede vem crescendo de maneira exponencial. Com isso, aumenta também a quantidade de ameaças e ataques às redes de computadores, não só às de grandes corporações como também às de pequenos negócios ou domésticas. Consequentemente, a habilidade para controlar tais redes especificando e aplicando políticas de segurança tem ganhado interesse na comunidade de pesquisa em redes, a qual vem buscando maneiras de garantir que essas políticas sejam corretamente aplicadas.

Essa habilidade vem sendo aprimorada com o surgimento de diferentes arquiteturas que apoiam a implantação de políticas de segurança. Tais políticas, em geral, são expressadas através de normas definidas por um analista de redes, que indicam as ações a serem tomadas para cada fluxo de dados presente na rede. Assim, há a necessidade de se operacionalizar essas normas e o mecanismo mais comum para fazê-lo atualmente é conhecido como *firewall*.

Um *firewall* protege uma rede de comunicação determinando os fluxos de dados que podem transitar através dele. Essa filtragem é realizada com base nas diferentes características do fluxos de dados presentes, como, por exemplo, a origem ou o destino dos dados sendo transmitidos. Para isso, um *firewall* é composto de regras que especificam quais dessas características são consideradas seguras ou não, refletindo as normas definidas pelo analista de redes.

Frequentemente, tais regras são especificadas por diferentes analistas em diferentes momentos e isso aumenta a chance do surgimento de conflitos entre elas. Por exemplo, uma nova regra, introduzida por um novo analista, pode acidentalmente permitir tráfego malicioso previamente bloqueado por outro analista. Tais conflitos causam *inconsistências* no comportamento do *firewall* e esse é um aspecto que vem, gradualmente, sendo alvo de novas pesquisas científicas.

Na literatura, a grande maioria de trabalhos atuais relacionados aos *firewalls* tem seu foco na otimização. Outro foco de pesquisa está na busca e na análise da consistência das regras que compõem o *firewall*, entretanto os trabalhos existentes não disponibilizam as implementações das propostas, portanto seus resultados não são reproduzíveis.

Esse artigo propõe o DETOX, uma implementação *open source*[1] de análise e busca de inconsistências na política de segurança implementada e validada em um *firewall* existente. As definições formais de inconsistências entre regras e o algoritmo aqui proposto foram baseados no artigo de [Al-Shaer and Hamed 2004]. Destaca-se como contribuições principais deste trabalho:

- Implementação e disponibilização do DETOX com seu código-fonte a fim de permitir a reproducibilidade dos testes e sua aplicação na prática.
- Correção de erros no algoritmo original de [Al-Shaer and Hamed 2004], além de sua extensão e adaptação para o DETOX.
- Aplicação do método de detecção de inconsistências em um *firewall* real.

Com relação ao terceiro ponto de contribuição, o DETOX foi aplicado ao *firewall* da rede acadêmica corporativa da Universidade Federal do Espírito Santo (UFES). Neste estudo foram descobertas inconsistências na configuração do *firewall* que eram previamente desconhecidas. Estas inconsistências foram apresentadas aos analistas de redes responsáveis para reparação, o que indica a aplicabilidade da ferramenta na prática em casos reais. Além disso, outro aspecto interessante a se destacar é o desempenho da ferramenta: a detecção das regras inconsistentes no caso de estudo (de tamanho considerável) leva apenas alguns milissegundos.

O restante do artigo está organizado como a seguir. A seção 2 fornece uma breve fundamentação teórica sobre inconsistências entre regras e a seção 3 indica trabalhos relacionados e justifica a necessidade da criação do DETOX. A seguir, a seção 3.1 apresenta uma análise aprofundada do algoritmo proposto por [Al-Shaer and Hamed 2004], indicando, em particular, discrepâncias encontradas no material original. A seção 4 descreve a implementação do DETOX, incluindo as correções para as discrepâncias citadas na seção anterior. A validação dessa implementação é discutida na seção 5.1. A aplicação do DETOX no caso de estudo real, o *firewall* da UFES, é apresentada na seção 5.2. Por fim, a seção 6 discorre sobre as conclusões e trabalhos futuros.

[1]Disponível em `https://github.com/llKieferll/DETOX`

2. Fundamentação Teórica

Um *firewall* é composto por regras que analisam e decidem se determinados fluxos de dados são considerados seguros ou não. Por sua vez, uma regra é composta por campos, onde cada campo refere-se a uma característica existente em cada fluxo de dados presente no *firewall* como, por exemplo, o IP de destino ou a porta de origem. Essa análise é realizada comparando cada campo de uma regra (o IP de origem definido na regra) à característica correspondente do fluxo sendo analisado (o IP de origem do fluxo). Caso ocorra um *match* entre todos os campos de uma regra referentes às características de um fluxo e o fluxo em si, é aplicada a ele a ação estabelecida por tal regra, normalmente presente em um campo denominado "Action". A existência desse campo é obrigatória e ele define se o fluxo definido pela regra é aceito ou bloqueado. Considere como exemplo uma regra fictícia a seguir, formada por três campos, IP de origem, IP de destino e *Action*:

$$10.0.0.1 \| * . * . * . * \| deny$$

Essa regra define que qualquer tráfego cuja origem é um IP específico (10.0.0.1) e cujo destino seja qualquer IP (*.*.*.*), deve ser negado (*deny*). Convém destacar que as regras são analisadas de forma sequencial, ou seja, uma regra anterior tem prioridade maior que uma regra posterior e apenas o primeiro *match* entre regra e fluxo é aplicado. Portanto, a inserção de uma nova regra como abaixo gera uma inconsistência pois a regra nova terá prioridade maior que a regra antiga e, como os campos referentes às características de fluxo são iguais, a regra antiga nunca será aplicada. Logo, o tráfego que antes era considerado malicioso agora será sempre aceito.

$$10.0.0.1 \| * . * . * . * \| accept$$
$$10.0.0.1 \| * . * . * . * \| deny$$

As subseções seguintes apresentam as definições das regras que compõem o *firewall* de exemplo a ser usado por este trabalho, bem como as definições das possíveis relações entre regras e as inconsistências que tais relações podem causar.

2.1. Relações entre regras

Nesta seção, apresentamos as definições das relações existentes entre duas regras. O *firewall* de exemplo de [Al-Shaer and Hamed 2004] foi também utilizado para facilitar a visualização de tais relações, desde o início da análise do algoritmo proposto, até a validação do DETOX. As regras que o compõem e seus campos encontram-se na Figura 1.

Considere que uma regra R possui n campos, para qualquer valor positivo de n. Utilizamos $R[i], 0 \leq i < n$ para indicar o campo i da regra R.

Definição 1: Duas regras R_x e R_y são **completamente disjuntas** (relação denotada como $R_x \Delta_{CD} R_y$) se:

$$\forall i : R_x[i] \not\bowtie R_y[i] \text{ onde } \bowtie \in \{\subset, \supset, =\}, i \neq \text{Action}.$$

Ou seja, duas regras são completamente disjuntas quando não há nenhuma relação de igualdade ou de interseção entre *todos* os campos de ambas regras.

		Source		Destination		
	Protocol	Address	Port	Address	Port	Action
1:	tcp,	140.192.37.20,	any,	*.*.*.*,	80,	deny
2:	tcp,	140.192.37.*,	any,	*.*.*.*,	80,	accept
3:	tcp,	*.*.*.*,	any,	161.120.33.40,	80,	accept
4:	tcp,	140.192.37.*,	any,	161.120.33.40,	80,	deny
5:	tcp,	140.192.37.30,	any,	*.*.*.*,	21,	deny
6:	tcp,	140.192.37.*,	any,	*.*.*.*,	21,	accept
7:	tcp,	140.192.37.*,	any,	161.120.33.40,	21,	accept
8:	tcp,	*.*.*.*,	any,	*.*.*.*,	any,	deny
9:	udp,	140.192.37.*,	any,	161.120.33.40,	53,	accept
10:	udp,	*.*.*.*,	any,	161.120.33.40,	53,	accept
11:	udp,	140.192.38.*,	any,	161.120.35.*,	any,	accept
12:	udp,	*.*.*.*,	any,	*.*.*.*,	any,	deny

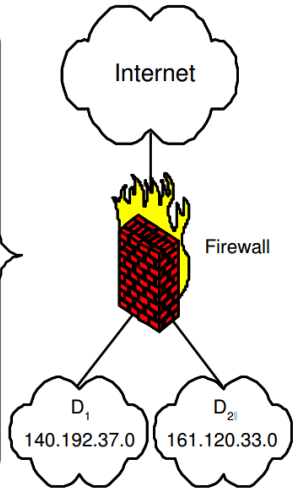


Figura 1. *Firewall* de exemplo de acordo com [Al-Shaer and Hamed 2004]

Definição 2: Duas regras R_x e R_y são **parcialmente disjuntas** ($R_x \Delta_{PD} R_y$) se:

$$\exists i, j \text{ tal que } R_x[i] \bowtie R_y[i] \text{ e } R_x[j] \not\bowtie R_y[j]$$
$$\text{onde } \bowtie \in \{\subset, \supset, =\}, i, j \neq \text{Action}, i \neq j.$$

Como exemplo de regras parcialmente disjuntas temos as regras 2 e 6 na Figura 1: todos os campos de ambas as regras coincidem, exceto o campo da porta de destino.

Definição 3: Duas regras R_x e R_y são um ***matching* exato** ($R_x \Delta_{EM} R_y$) se:

$$\forall i : R_x[i] = R_y[i] \text{ onde } i \neq \text{Action}.$$

Ou seja, duas regras possuem um *matching* exato quando todos os campos exceto o campo *Action* forem iguais.

Definição 4: Duas regras R_x e R_y são um ***matching* inclusivo** ($R_x \Delta_{IM} R_y$) se:

$$\forall i : R_x[i] \subseteq R_y[i] \text{ e } \exists j \text{ tal que } R_x[j] \neq R_y[j] \text{ onde } i, j \neq \text{Action}.$$

Como exemplo de regras com *matching* inclusivo temos as regras 1 e 2 na Figura 1.Vale destacar que o quantificador existencial na definição acima é necessário para garantir que o *matching* não é exato.

Definição 5: Duas regras R_x e R_y são **correlacionadas** ($R_x \Delta_C R_y$) se:

$$\forall i : R_x[i] \bowtie R_y[i] \text{ e } \exists j, k \text{ tal que } R_x[j] \subset R_y[j] \text{ e } R_x[k] \supset R_y[k]$$
$$\text{onde } \bowtie \in \{\subset, \supset, =\}, i, j, k \neq \text{Action}, j \neq k.$$

Como exemplo de regras correlacionadas temos as regras 1 e 3 na Figura 1, pois os campos de endereço de origem e destino estão contidos um no outro nessas regras.

2.2. Inconsistências

Baseado nas relações apresentadas na seção anterior, definimos nesta seção as possíveis *inconsistências* causadas por essas relações. Tais definições também são apresentadas

no trabalho de [Al-Shaer and Hamed 2004]. Considere que, para uma regra R qualquer, utilizamos $R[Action]$, para indicar o campo referente à ação a ser tomada pela regra R (aceitar ou bloquear) em relação ao fluxo também definido pela regra R. Assim, temos:

Shadowing: Uma regra R_y é sombreada por uma regra de prioridade maior R_x se uma das seguintes condições é satisfeita:

$$R_x \Delta_{EM} R_y, R_x[Action] \neq R_y[Action] \quad (1)$$
$$R_y \Delta_{IM} R_x, R_x[Action] \neq R_y[Action] \quad (2)$$

Exemplo: regras 4 e 3 na Figura 1. A regra 4 possui uma relação de *matching* inclusivo com a regra 3, que possui prioridade maior.

Correlation: Duas regras R_x e R_y causam uma inconsistência de correlação se a seguinte condição é satisfeita:

$$R_x \Delta_C R_y, R_x[Action] \neq R_y[Action] \quad (3)$$

Exemplo: regras 1 e 3 na Figura 1.

Generalization: Uma regra R_y é uma generalização de uma regra de prioridade maior R_x se a seguinte condição é satisfeita:

$$R_x \Delta_{IM} R_y, R_x[Action] \neq R_y[Action] \quad (4)$$

Exemplo: regras 2 e 1 na Figura 1, onde a regra 2 generaliza a regra 1.

Redundancy: Uma regra R_y é redundante à uma regra de prioridade maior R_x se uma das seguintes condições é satisfeita:

$$R_x \Delta_{EM} R_y, R_x[Action] = R_y[Action] \quad (5)$$
$$R_y \Delta_{IM} R_x, R_x[Action] = R_y[Action] \quad (6)$$

Exemplo: regras 7 e 6 na Figura 1, onde a regra 7 é redundante em relação à regra 6.

Além disso, uma regra R_x é redundante à uma regra de prioridade menor R_y se a seguinte condição é satisfeita:

$$\begin{gathered} R_x \Delta_{IM} R_y, R_x[Action] = R_y[Action] \\ \text{e } \nexists R_z \text{ tal que } prioridade(R_x) > prioridade(R_z) > prioridade(R_y), \\ R_x\{\Delta_{IM}, \Delta_C\} R_z, R_x[Action] \neq R_z[Action] \end{gathered} \quad (7)$$

Exemplo: regras 4 e 8 na Figura 1. Convém destacar que esta última condição (7) é apresentada, mas não implementada no artigo original de [Al-Shaer and Hamed 2004].

3. Trabalhos Relacionados

Nos trabalhos que têm seu foco na análise e deteção de inconsistências, [Eppstein and Muthukrishnan 2001] e [Hari et al. 2000] fazem uma análise limitando-se apenas à um tipo de inconsistência existente, a de *correlação*. Já os trabalhos de [Bartal et al. 1999] e [Mayer et al. 2000] propõem uma política em linguagem de alto nível para definir, analisar e em seguida mapear novas regras de filtragem, mas como

a maior parte dos *firewalls* utilizados hoje em dia contém regras de baixo nível, redefinir políticas já existentes em linguagens de alto nível torna-se um esforço muito grande frente ao benefício conquistado. Além disso, as ferramentas para análise propostas nestes artigos ("*Firmato*" e "*Fang*", respectivamente) não parecem estar mais disponíveis.

Já [Khorchani et al. 2012] propõem o uso de *model checking* para a análise e detecção de inconsistências usando *Visibility Logic (VL)*, um tipo de lógica multi-modal. Os autores argumentam que as definições de inconsistências apresentadas na seção 2.2 podem ser vistas como casos particulares de fórmulas em VL, e descrevem a implementação de um *model checker* dedicado para VL. Infelizmente, esta implementação também não está disponível. Mais recentemente, [Mukkapati and Ch.V.Bhargavi 2013] realizam um trabalho interessante utilizando álgebra relacional para a detecção de inconsistências, definindo o que são e como podem ser encontradas, juntamente com a ideia de combinação de regras similares a fim de diminuir seu número total e otimizar o *firewall*. Entretanto, o trabalho também não apresenta um algoritmo ou uma ferramenta.

Merece destaque o trabalho de [Yuan et al. 2006], que fornece uma pesquisa mais completa, apresentando regras cujas ações aplicáveis sobre os fluxos de dados que vão além de *accept* e *deny* e analisando a relação existente entre três ou mais regras, fornecendo assim uma perspectiva de trabalhos futuros para o DETOX. Entretanto, novamente, o *toolkit* "Fireman" proposto não foi encontrado, nem qualquer código relacionado a ele, o quê inviabiliza o seu uso. Esta é a mesma situação da ferramenta original criada por [Al-Shaer and Hamed 2004], chamada de *Firewall Policy Advisor* (FPA).

Um vez que nenhuma das ferramentas similares propostas na literatura se encontram mais disponíveis, buscou-se o desenvolvimento do DETOX para preencher esta lacuna. Convém destacar que a indisponibilidade das demais ferramentas impede um estudo comparativo entre o DETOX e as demais.

3.1. Discrepâncias do algoritmo original de [Al-Shaer and Hamed 2004]

A base do trabalho aqui descrito iniciou-se com a análise do algoritmo em pseudocódigo apresentado por [Al-Shaer and Hamed 2004]. Imediatamente percebe-se quatro discrepâncias entre o algoritmo sugerido e os resultados apresentados no artigo [Al-Shaer and Hamed 2004]. A condição (7) da seção anterior não foi implementada no algoritmo sugerido. Na verdade, caso a primeira linha da condição (7) seja satisfeita, o caso é imediatamente considerado como *Shadowing* sem análise das condições seguintes. Além disso, a condição (2) é considerada como *Generalization* quando trata-se de um caso de *Shadowing* e a condição (4) é considerada como *Shadowing* quando trata-se de um caso de *Generalization*.

Apesar disso, os resultados finais em [Al-Shaer and Hamed 2004] aparentavam estar corretos e não condiziam com o algoritmo sugerido. Decidiu-se então realizar uma análise manual do *firewall* com base nas definições da seção anterior a fim de validar os resultados finais. Por fim, todos eles estavam corretos, exceto por uma redundância entre as regras 4 e 8, a qual não foi detectada e também uma generalização entre as regras 12 e 11, embora essa última possa não ter sido documentada devido à regra 11 ser um caso de inconsistência de irrelevância. Entretanto, os resultados apresentados pelo algoritmo sugerido estavam, em sua grande maioria, incorretos. Devido às discrepâncias encontradas, decidiu-se implementar o DETOX para corrigi-las, como descrito a seguir.

4. Desenvolvimento

O funcionamento e uso do DETOX pode ser dividido em quatro etapas. A primeira etapa refere-se ao preenchimento de um arquivo de configuração que reflete as configurações do *firewall* a ser analisado, necessários para o funcionamento do DETOX e explicado na subseção 4.1. A subseção 4.2 lida com a leitura das regras, sendo esta realizada de acordo com os dados fornecidos pelo arquivo de configuração devidamente preenchido. Em seguida, o tratamento necessário é aplicado às regras compostas encontradas transformando-as em regras simples, o que será explicado na seção 4.3. Por fim, na seção 4.4 apresentamos o algoritmo de busca de inconsistências utilizado pelo DETOX. Maiores detalhes sobre os itens citados em cada subseção seguinte podem ser encontrados no site[2] da ferramenta.

4.1. Arquivo de Configuração

O arquivo de configuração deve ser preenchido pelo usuário e conter informações básicas para o funcionamento do DETOX. As informações mais básicas incluem o nome de cada campo existente em uma regra e quais destes campos serão incluídos na busca por inconsistências. Além disso, diferentes campos geralmente são comparados de diferentes maneiras. Tome como exemplo endereços de rede e interfaces. Endereços podem ser diferentes, iguais ou ainda estarem contidos um no outro. Já interfaces podem apenas ser diferentes ou iguais.

Por isso, é necessário que seja fornecido também um método para comparar cada par de campos correspondentes entre duas regras para que seja possível descobrir a relação entre eles (tais métodos não são necessáriamente distintos). Para fornecer maior facilidade de uso, quatro métodos iniciais que abrangem a maior parte dos campos mais utilizados em uma regra já constam como métodos *default* no arquivo. Com a ferramenta sendo implementada em Java, todos esses dados foram organizados em estruturas do tipo *enums*, a fim de promover a modularização dos dados e a flexibilidade necessária para adaptar o arquivo a qualquer tipo de *firewall* ou qualquer outro ambiente que utilize regras em formato similar.

4.2. Leitura das regras

Com o arquivo de configuração definido, a leitura das regras do *firewall* em questão pode ser realizada. Ressaltamos que tais regras podem conter um campo, não obrigatório, cuja função é unicamente definir se encontram-se habilitadas ou desabilitadas no *firewall*; esse campo é chamado de "*Enabled*". As regras também devem conter, obrigatoriamente, o campo "*Action*", apresentado na seção 2. Portanto, deve-se verificar previamente a existência de ambos a fim de garantir que a estrutura definida no arquivo de configuração está correta, bem como garantir que somente regras pertinentes serão lidas.

Após essa verificação, seguida da leitura, tem-se uma lista contendo as regras ativas, as quais, nessa lista, são compostas apenas pelos campos que devem ser analisados na busca por inconsistências, como definido no arquivo de configuração explicado na seção anterior. A estrutura de lista permite manter a ordem (prioridade) das regras fornecendo um acesso imediato a elas. No entanto, os campos podem conter dois ou mais valores distintos e agrupados, geralmente separados por vírgula. Tais campos são chamados compostos e devem ser transformados em campos simples, como é explicado a seguir.

[2]Disponível em `https://github.com/llKieferll/DETOX`

4.3. Simplificação de regras

Os campos de uma regra podem possuir um único elemento representando um valor, um único elemento representando uma faixa de valores ou ainda dois ou mais elementos distintos, podendo representarem tanto valores únicos como também faixas de valores. Uma regra onde todos os seus campos contém um único elemento (mesmo que tal elemento seja uma faixa de valores) é denominada *regra simples*. Tome como exemplo, regras formadas por três campos, IP de origem, IP de destino e porta, nessa ordem. As seguintes regras são exemplos de regras simples:

1. 10.0.0.1 || 10.0.1.1 || 50
2. 10.0.0.1 || 10.0.1.1 || 51
3. 10.0.0.1 || 10.0.1.1 || 50-55
4. 10.0.0.* || 10.0.1.* || *

Note que, nos exemplos citados acima, o campo referente à porta na regra 3, *"50-55"*, é considerado simples pois é um elemento único que representa uma faixa de valores. Todos os campos das regras citadas possuem um único elemento tornando-as regras simples. No caso de uma regra conter dois ou mais elementos, teremos diferentes regras condensadas em apenas uma. Esse tipo de regra é denominada *regra composta*. Seguindo o mesmo padrão anterior, são exemplos de regras compostas:

5. 10.0.0.1 || 10.0.1.1 || 50, 51
6. 10.0.0.* || 10.0.1.*, 10.0.2.* || *
7. 10.0.0.1, 10.0.0.2 || 10.0.1.1, 10.0.1.2|| 50

Note que o campo referente à porta na regra 5 possui dois valores distintos (50 e 51) e essa regra deve ser simplificada. Ao fazê-lo duas novas regras são geradas, uma para cada elemento distinto, para cada campo composto. Os campos simples são repetidos e os elementos distintos são distribuídos em suas respectivas posições nas regras geradas, consequentemente levando às regras simples 1 e 2. De maneira similar, a simplificação da regra 6 leva às seguintes regras simples:

- 10.0.0.* || 10.0.1.* || *
- 10.0.0.* || 10.0.2.* || *

E a simplificação da regra 7 leva às seguintes regras simples:

- 10.0.0.1 || 10.0.1.1 || 50
- 10.0.0.1 || 10.0.1.2 || 50
- 10.0.0.2 || 10.0.1.1 || 50
- 10.0.0.2 || 10.0.1.2 || 50

Com base nesse conhecimento, é realizada a simplificação das regras compostas. Analisamos regra por regra, em ordem, verificando os campos de cada uma e, caso sejam todos simples, a armazenamos em uma estrutura, como citado na seção anterior, destinada a conter apenas regras simples. Caso uma regra composta seja encontrada, devemos simplificá-la. Todas as regras simples geradas são transferidas juntas para a lista de regras a fim de ocuparem conjuntamente a posição da regra composta que as gerou. Assim, todas as regras geradas têm a mesma prioridade da regra que as gerou.

4.4. Busca por inconsistências

Com a certeza que todas as regras compostas foram transformadas em regras simples, inicia-se então a busca por inconsistências entre elas. O algoritmo 1 representa a versão final que foi implementada no DETOX, incluindo as adaptações que corrigem as discrepâncias citadas na seção 3.

Algorithm 1 Inconsistency Finder

```
1:  procedure FIND
2:      Input: rules - ArrayList
3:      Output: incons_list - HashMap

4:      active_fields ← Lista de campos ativos definidos no arquivo de configuração
5:      incons_list ← {}
6:      Remover "Enabled" de active_fields
7:      Remover "Action" de active_fields
8:      for Cada R_x em rules do
9:          for Cada R_y em rules onde prioridade(R_x) > prioridade(R_y) do
10:             rules_relation ← NONE
11:             for Cada field em active_fields do
12:                 fields_relation ← field.Compare(R_x[field], R_y[field])
13:                 rules_relation ← Rel(rules_relation, fields_relation)
14:             if rules_relation ≠ DISJOINT then
15:                 switch rules_relation
16:                     case CORRELATED do
17:                         if R_x[Action] ≠ R_y[Action] then
18:                             Adicionar (R_x, CORRELATION, R_y) em incons_list
19:                     case SUPERSET do
20:                         if R_x[Action] = R_y[Action] then
21:                             Adicionar (R_y, REDUNDANCY, R_x) em incons_list
22:                         else
23:                             Adicionar (R_y, SHADOWING, R_x) em incons_list
24:                     case EXACT do
25:                         if R_x[Action] = R_y[Action] then
26:                             Adicionar (R_y, REDUNDANCY, R_x) em incons_list
27:                         else
28:                             Adicionar (R_y, SHADOWING, R_x) em incons_list
29:                     case SUBSET do
30:                         if R_x[Action] = R_y[Action] then
31:                             XZ_fields_rel ← NONE
32:                             XZ_rules_rel ← NONE
33:                             Z_exists ← false
34:                             for Cada regra R_z entre R_x e R_y do
35:                                 for Cada field em active_fields do
36:                                     XZ_fields_rel ← field.Compare(R_x[field],R_z[field])
37:                                     XZ_rules_rel ← Rel(XZ_rules_rel, XZ_fields_rel)
38:                                 if rules_relation ∈ CORRELATED,SUBSET then
39:                                     if R_x[Action] ≠ R_z[Action] then
40:                                         Z_exists ← true
41:                             if Not Z_exists then
42:                                 Adicionar (R_x, REDUNDANCY, R_y) em incons_list
43:                         else
44:                             Adicionar (R_y, GENERALIZATION, R_x) em incons_list
```

Inicialmente, removemos os campos *Enabled*, pois não tem influência na relação entre regras, e *Action*, pois influencia apenas o tipo de inconsistência, caso exista, decorrente da relação encontrada entre as regras. Devemos então analisar os pares de regras R_x e R_y, onde a prioridade de R_y é sempre menor que a prioridade de R_x.

A relação momentânea inicial entre duas regras é inexistente. A cada comparação de campos correspondentes de duas regras sendo analisadas (linha 12 do Algoritmo 1), atualizamos a relação momentânea entre as regras em questão, com base na combinação entre a relação momentânea existente e o resultado da comparação entre os campos analisados (linha 13 do Algoritmo 1) através do Algoritmo 2. Cada nova comparação de campos subsequentes irá atualizar a relação momentânea até que todos os campos das regras sendo analisadas sejam comparados e a relação final entre tais regras seja encontrada.

No Algoritmo 2, realizamos a atualização da relação momentânea entre duas regras com base na ultima comparação entre campos realizada e a relação momentânea atual entre elas. Para tal, verificamos qual a relação encontrada entre entre tais campos (linha 4 do Algoritmo 2). Em seguida, verificamos a relação momentânea atual entre as regras (linhas 6, 8, 11, 13, 16 e 18 do Algoritmo 2). A combinação dessas duas informações nos fornece uma nova relação momentânea entre as regras sendo analisadas.

Algorithm 2 Rules Relation Finder

```
 1: procedure REL
 2:     Input: rules_relation - Int, field_relation - Int
 3:     Output: <Relation between rules> - Int

 4:     switch field_relation
 5:         case EQUALS do
 6:             if rules_relation = NONE then
 7:                 return EXACT
 8:             else
 9:                 return rules_relation
10:         case CONTAINS do
11:             if rules_relation = {SUBSET,CORRELATED} then
12:                 return CORRELATED
13:             else if rules_relation ≠ DISJOINT then
14:                 return SUPERSET
15:         case CONTAINED do
16:             if rules_relation = {SUPERSET,CORRELATED} then
17:                 return CORRELATED
18:             else if rules_relation ≠ DISJOINT then
19:                 return SUBSET
20:         case default do
21:             return DISJOINT
```

Após todos os campos do par de regras em questão serem analisados, encontraremos a relação final entre as regras. Caso sejam disjuntas (linha 14) não há inconsistências. Caso contrário, devemos analisar sua relação final (linha 15 do Algoritmo 1) e os campos *Action* de ambas as regras (linhas 17, 20, 25, 30 e 39 do Algoritmo 1). Com isso, encontramos o tipo de inconsistência existente entre elas, a qual deve ser armazenada na lista de inconsistências. Terminado o procedimento para todos os pares de regras possíveis, temos como resultado uma *HashMap* que contém a todas as inconsistências encontradas.

5. Testes realizados

Com o novo algoritmo implementado, realizamos a sua validação e, após esta, sua aplicação em um caso de estudo real. Para a validação, utilizamos o *firewall* de exemplo de [Al-Shaer and Hamed 2004], exibido na Figura 1, a fim de replicar os resultados esperados de acordo com teste manual realizado na seção 3. Como caso de estudo real, utilizamos a rede acadêmica da UFES, como descrito na seção 1. As subseções seguintes tratam da validação e da aplicação em caso de estudo real.

5.1. Validação

Para a validação do DETOX, realizamos uma comparação entre o resultado fornecido pela versão final do nosso algoritmo, com os resultados encontrados durante nosso teste manual, como realizado na seção 3. Os resultados apresentados pelo DETOX foram exatamente os resultados esperados, sendo iguais aos resultados expostos pelo teste manual e realizando duas detecções adicionais em relação aos resultados obtidos por [Al-Shaer and Hamed 2004], sendo uma *redundancy* e uma *generalization* entre as regras 4 e 8 e as regras 12 e 11, respectivamente, da Figura 1. Os resultados pertinentes são apresentados como na Tabela 3, omitindo as regras que não apresentam inconsistências.

Tabela 1. Inconsistências no *firewall* exemplo

Rule No	CORRELATED	GENERALIZATION	REDUNDANT	SHADOWED
1	{3}			
2		{1}		
4			{8}	{2, 3}
5	{7}			
6		{5}		
7			{6}	
8		{2, 3, 6, 7}		
9			{10}	
12		{9, 10, 11}		

5.2. Caso de estudo

Após realizada a validação do DETOX, buscamos aplicá-lo em um caso de estudo real, a fim de obter resultados também reais e verificar sua viabilidade. Para isso, utilizamos o *firewall* do Núcleo de Tecnologia de Informação (NTI) da UFES. Verificamos quais campos deveriam ser analisados na busca por inconsistências e quais campos poderiam ser ignorados e alteramos o arquivo de configuração para refletir os campos que compõem o *firewall*. Os métodos *default* de comparação foram suficientes para abranger todos os campos inclusos na busca por inconsistências.

A definição desse *firewall* contém 207 regras, todas contendo o campo *Enabled* e sendo várias delas compostas. Dessas 207 regras, 174 estavam habilitadas. Das habilitadas, após a simplificação conforme descrito na seção 4.3, obtemos um total de 317 regras simples. Porém, muitas delas continham *aliases* (um nome dado a uma rede ou endereço

de IP ao invés do endereço propriamente dito). Por questões de anonimidade, tais *aliases* foram substituídos por IPs e redes fictícios, mas de modo a manter quaisquer relações existentes entre os IPs reais sem comprometer os resultados. Os resultados obtidos através da aplicação do DETOX encontram-se na Tabela 2:

Tabela 2. Inconsistências no *firewall* do NTI

Rule No	CORRELATED	GENERALIZATION	REDUNDANT
12			{46}
13			{21,46}
14			{25}
15			{24}
18	{22,23,42,45,55,57}		
19			{46}
20			{46}
21			{46}
31			{52}
35			{54}
41			{46}
48			{46}
60			{24}
61			{25}
72	{301}		
81			{71}
82			{71}
83			{71}
84			{71}
85			{125}
125	{301}		
187		{186}	
188		{184}	
189		{185}	
289	{301}		
Total de Inconsistências	**9**	**3**	**19**
Regras Inconsistentes	**11 de 317**	**6 de 317**	**25 de 317**
Porcentagem de Inconsistências	**3.47%**	**1.89%**	**7.88%**

Observe que não há uma coluna referente à inconsistência do tipo *shadowing*. Isso ocorre porque não foram encontradas inconsistências desse tipo, portanto, a omitimos nos resultados. Encontramos que, das 317 regras simples, 41 regras distintas apresentaram algum tipo de inconsistência, o que representa cerca de 12,93% do total de regras existentes no *firewall*. É fácil notar que, como citado na seção 1, mesmo um *firewall* administrado por pessoas experientes e competentes pode ser comprometido sem o suporte adequado para garantir a consistência da política de segurança aplicada pelas regras de um *firewall*.

Para exemplificar o caso real, exibimos aqui a composição da regra 18 e suas inconsistências de *correlation*, a regra 81 e sua inconsistência de *redundancy* e a regra 187 e sua inconsistência de *generalization*. Os campos *hits*, *logging*, *description* e *time* não foram incluídos na análise pois referenciavam características meramente informativas.

Tabela 3. Exemplo de regras inconsistentes no *firewall* real

ID Pos-Simp	Interface	Source	Destination	Service	Action	ID Pre-Simp
18	inside	any	10.10.2.*	ip	deny	10
22	inside	10.20.3.*	any	ip	permit	13
23	inside	10.20.4.*	any	ip	permit	13
42	inside	10.20.5.21	any	ip	permit	16
45	inside	10.20.6.1-11	any	ip	permit	19
55	inside	10.20.7.20	any	ip	permit	28
57	inside	10.20.6.30	any	ip	permit	30
71	outside	any	any	icmp	permit	40
81	outside	10.20.4.40	any	icmp	permit	50
186	outside	100.200.50.60	100.250.10.10-20	tcp/smtp	deny	112
187	outside	any	100.250.10.10-20	tcp/smtp	permit	113

Como visto anteriormente, as regras passam, inicialmente, por um processo de simplificação. As colunas ”ID Pre-Simp”e ”ID Pos-Simp”indicam, respectivamente, o número (prioridade) da regra em questão antes e depois do processo de simplificação. Por exemplo, a primeira regra da lista encontrava-se a posição 10 antes da simplificação e encontra-se na posição 18 após a simplificação. Por questões de confidenciabilidade, não exibimos os IPs reais utilizados na universidade e trocamos seus números reais, com o cuidado de não alterar suas relações.

6. Conclusões e trabalhos futuros

Este trabalho realizou uma análise do algoritmo de detecção de inconsistências proposto por [Al-Shaer and Hamed 2004]. Para tal, implementamos o algoritmo fornecido em pseudo-código e verificamos sua validade utilizando o *firewall* de exemplo, sendo ambos fornecidos por [Al-Shaer and Hamed 2004].

Discrepâncias entre os resultados do algoritmo e os resultados exibidos foram encontradas. Assim, implementamos nosso próprio algoritmo na ferramenta *open source* DETOX, de maneira a incluir adaptações para corrigir tais discrepâncias e o validamos utilizando o mesmo *firewall* de exemplo usado por [Al-Shaer and Hamed 2004] chegando assim aos resultados esperados.

A viabilidade do novo algoritmo foi testada em uma *firewall* real, onde 307 regras simples foram analisadas e inconsistências foram detectadas, com todo o processo ocor-

rendo em um tempo abaixo de 300ms. Os resultados foram devidamente apresentados ao administrador de rede responsável.

Como trabalhos futuros, pretendemos modificar o DETOX para lidar com múltiplos *firewalls*, bem como adaptá-lo para ser aplicado em redes definidas por *software* (SDN), sobre protocolos como *OpenFlow* e *Border Gateway Protocol* (BGP).

Mais ainda, inspirados pelo trabalho de [Yuan et al. 2006], pretendemos trazer para nossa ferramenta a identificação de quaisquer tipos de inconsistências que possam existir entre combinações de n regras, não somente para *firewalls*, mas também para os ambientes citados acima. Atualmente cada par é analisado de maneira independente pela ferramenta, mas, no caso de um conjunto de regras gerar uma inconsistência com uma única regra ou um outro conjunto de regras, cada um dos pares possíveis desses conjuntos é reportado como um caso de inconsistência. Tal análise pode gerar indicativos de inconsistências ainda mais elaborados e planeja-se incorporá-la ao DETOX em breve.

Além disso, há a possibilidade de adaptar nossa ferramenta para funcionamento *on-the-fly*, constantemente observando as regras existentes no ambiente a ser analisado. Acreditamos que, para redes SDN o maior desafio será a viabilidade devido ao grande volume de mudanças ocorrendo nas redes.

Referências

Al-Shaer, E. and Hamed, H. (2004). Modeling and management of firewall policies. *Network and Service Management, IEEE Transactions on*, 1(1):2–10.

Bartal, Y., Mayer, A., Nissim, K., and Wool, A. (1999). Firmato: a novel firewall management toolkit. In *Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on*, pages 17–31.

Eppstein, D. and Muthukrishnan, S. (2001). Internet packet filter management and rectangle geometry. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '01, pages 827–835, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

Hari, A., Suri, S., and Parulkar, G. (2000). Detecting and resolving packet filter conflicts. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1203–1212 vol.3.

Khorchani, B., Halle, S., and Villemaire, R. (2012). Firewall anomaly detection with a model checker for visibility logic. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pages 466–469.

Mayer, A., Wool, A., and Ziskind, E. (2000). Fang: a firewall analysis engine. In *Security and Privacy, 2000. S P 2000. Proceedings. 2000 IEEE Symposium on*, pages 177–187.

Mukkapati, N. and Ch.V.Bhargavi (2013). Detecting policy anomalies in firewalls by relational algebra and raining 2d-box model. *International Journal of Computer Science and Network Security, IJCSNS*, 13(5).

Yuan, L., Chen, H., Mai, J., Chuah, C.-N., Su, Z., and Mohapatra, P. (2006). Fireman: a toolkit for firewall modeling and analysis. In *Security and Privacy, 2006 IEEE Symposium on*, pages 15 pp.–213.

WTF 2016
Sessão Técnica 3
Algoritmos Distribuídos

Implementando Diversidade em Replicação Máquina de Estados

Caio Yuri Silva Costa[1] **and Eduardo Adilio Pelinson Alchieri**[1]

[1]Departamento de Ciência da Computação
Universidade de Brasília

caio.costa@outlook.com,alchieri@unb.br

Abstract. *The security properties of a system could be impaired by an opponent that exploits its vulnerabilities. An alternative to mitigate this risk is the implementation of intrusion-tolerant systems. State Machine Replication (SMR) is widely used in these implementations. However, the proposed solutions do not allow replica implementation diversity, consequently, the same attack could compromise all the system. In this way, this work proposes an architecture to allow diversity in SMR implementations and shows how this architecture was integrated in the* BFT-SMaRt. *A set of experiments shows the practical behavior of the proposed solutions.*

Resumo. *Vulnerabilidades podem comprometer as propriedades de segurança de um sistema quando adequadamente exploradas por um atacante. Uma alternativa para mitigar este risco é a implementação de sistemas tolerantes a intrusões. Uma abordagem muito utilizada para estas implementações é a replicação Máquina de Estados (RME). Porém, as soluções existentes não suportam diversidade na implementação das réplicas, de forma que um mesmo ataque pode comprometer todo o sistema. Neste sentido, este trabalho propõe uma arquitetura para fornecer suporte à diversidade de implementação em RMEs e mostra como a mesma foi integrada no* BFT-SMaRt. *Um conjunto de experimentos mostra o comportamento prático das soluções propostas.*

1. Introdução

Vulnerabilidades podem comprometer as propriedades de segurança de um sistema caso sejam adequadamente exploradas por um atacante. Em vista disso, os sistemas devem ser construídos de forma a tolerarem intrusões [Fraga and Powell 1985, Veríssimo et al. 2003]: o sistema deve permanecer funcionando corretamente mesmo que uma parte do mesmo seja invadida e controlada por um atacante. Funcionar corretamente significa manter suas propriedades, como disponibilidade, integridade e confidencialidade [Avizienis et al. 2004]. Outro fator que pode comprometer estes sistemas são *bugs* de *software* ou *hardware*, desta forma é necessário que estes sistemas também funcionem corretamente caso alguma parte de mesmo esteja produzindo dados incorretos.

Uma abordagem amplamente utilizada na implementação de sistemas tolerantes a falhas, tanto por *crash* [Schneider 1990] quanto Bizantinas [Castro and Liskov 2002] é a Replicação Máquina de Estados (RME) [Schneider 1990]: esta abordagem consiste em replicar os servidores (permitindo que alguns deles falhem, i.e., sejam invadidos e controlados por um atacante ou apresentem algum *bug* de *software* ou *hardware*) e coordenar

as interações entre os clientes e as réplicas dos servidores, com o intuito de que as várias réplicas apresentem a mesma evolução em seus estados.

Embora existam muitas propostas de sistemas e protocolos para RME (ex.: [Amir et al. 2011, Castro and Liskov 2002, Clement et al. 2009b, Clement et al. 2009a, Guerraoui et al. 2010, Kotla et al. 2009, Veronese et al. 2013, Bessani et al. 2014], para citar apenas alguns), nenhum deles fornece suporte para implementação de diversidade nas réplicas [Obelheiro et al. 2005, Bessani et al. 2009, Garcia et al. 2011, Platania et al. 2014, Garcia et al. 2014], i.e., implementar diferentes réplicas em diferentes linguagens de programação aumentando o grau de independência de falhas (duas ou mais réplicas não falham pelo mesmo motivo). Sendo assim, nestes sistemas existentes, um atacante pode utilizar o mesmo ataque para comprometer todas as réplicas do sistema ou ainda, como o mesmo *software* está executando nas diversas réplicas, um mesmo *bug* pode comprometer toda a aplicação.

Com o objetivo de preencher esta lacuna, este trabalho apresenta nossos esforços para adicionar suporte a diversidade na implementação das réplicas (e clientes) no BFT-SMaRt [Bessani et al. 2014], que é uma biblioteca escrita em Java para implementação de uma RME tolerante a falhas Bizantinas. Nosso objetivo não é implementar esta biblioteca em diferentes linguagens, mas sim fornecer interfaces para que clientes e réplicas possam ser desenvolvidas em diferentes linguagens. A implementação de uma biblioteca completa para RME demanda um grande esforço financeiro e muito tempo. Por exemplo, demorou mais de 5 anos para tornar o Bft-SMaRt um sistema robusto [Bessani et al. 2014]. Este trabalho busca propor uma alternativa mais viável.

A arquitetura proposta utiliza o conceito de estado abstrato independente da linguagem [Castro et al. 2003] e apresenta pelo menos dois grandes desafios: *(1)* internamente em um processo – possibilitar a comunicação entre diferentes linguagens (ex.: uma réplica escrita em C precisa executar métodos do BFT-SMaRt escritos em Java e vice-versa); e *(2)* entre processos – possibilitar a troca de informações entre réplicas (ou entre réplicas e clientes) implementadas em linguagens diferentes (ex.: a representação de um vetor em C é diferente da representação em Java).

Resumidamente, as principais contribuições deste trabalho são: *(i)* proposta de uma arquitetura para suportar diversidade na implementação de réplicas (e clientes) de uma RME e descrição de como a mesma foi integrada no Bft-SMaRt, *(ii)* discussão de como o ambiente de execução deve ser configurado para aumentar a segurança das aplicações, e *(iii)* apresentação e análise de uma série de experimentos com as implementações realizadas, trazendo uma melhor compreensão a respeito do funcionamento de diversidade em uma RME.

O restante deste texto está organizado da seguinte forma. A Seção 2 discute mais detalhadamente o conceito de diversidade, enquanto que a Seção 3 apresenta a biblioteca para RME (Bft-SMaRt) utilizada neste trabalho. A Seção 4 discute nossa proposta para uso de diversidade no Bft-SMaRt. Vários experimentos são discutidos na Seção 5. Finalmente, a Seção 6 apresenta as conclusões do trabalho bem como os trabalhos futuros.

2. Diversidade

A abordagem de usar diferentes implementações de um sistema (ou parte dele) para tolerar falhas de *software* foi proposta ainda na década de 70 [Randell 1975,

Avizienis and Chen 1977]. A ideia central destas propostas é que implementações diferentes não apresentem as mesmas falhas de *software*. Recentemente, vários trabalhos abordaram (apenas de forma teórica) o uso de diversidade para prover independência de falhas e assim tolerar intrusões [Obelheiro et al. 2005, Bessani et al. 2009, Garcia et al. 2011, Platania et al. 2014, Garcia et al. 2014].

Existem vários pontos de um sistema (ou parte dele) que podem ser diversificados, formando os seguintes eixos de diversidade [Obelheiro et al. 2005]:

1. Implementação: Consiste na implementação de diferentes versões de um *software*.
2. Administração: Consiste na distribuição dos componentes de um sistema em diferentes domínios administrativos.
3. Localização: Consiste em espalhar os vários componentes físicos de um sistema entre diferentes sítios de instalação.
4. COTS (*Commercial Off The Shelf*): Consiste em utilizar vários COTS diferentes que implementam algumas funcionalidades, como compiladores, bibliotecas, etc.
5. Sistema Operacional: Consiste na distribuição da aplicação entre diferentes sistemas operacionas.
6. Métodos: Consiste na utilização de métodos distintos para garantir algum atributo de segurança, como por exemplo cifrar sucessivamente um dado com mais de um algoritmo criptográfico.
7. *Hardware*: Consiste na distribuição da aplicação entre diferentes *hardwares*.

Este trabalho propõe uma arquitetura que possibilita o emprego de diferentes linguagens de programação na implementação de diferentes réplicas de uma RME. Desta forma, a arquitetura proposta fornece suporte principalmente para diversidade de implementação em RMEs, embora seja possível explorar outros eixos de diversidade, como por exemplo o sistema operacional e o *hardware* (Seção 5.2).

3. BFT-SMART: Implementação de Replicação Máquina de Estados

Em uma replicação Máquina de Estados [Schneider 1990], as réplicas devem apresentar a mesma evolução em seus estados: *(i)* partindo de um mesmo estado e *(ii)* executando o mesmo conjunto de requisições na mesma ordem, *(iii)* todas as réplicas devem chegar ao mesmo estado final, definindo o determinismo de réplicas. Para prover *(i)*, basta iniciar todas as réplicas com o mesmo estado (i.e., iniciar todas as variáveis que representam o estado com os mesmos valores nas diversas réplicas). Garantir o item *(ii)* envolve a utilização de um protocolo de difusão atômica. O problema da *difusão atômica* [Hadzilacos and Toueg 1994] (ou difusão com ordem total) consiste em fazer com que todos os processos corretos de um grupo entreguem todas as mensagens difundidas neste grupo na mesma ordem. Já para prover *(iii)* é necessário que as operações executadas pelas réplicas sejam determinísticas (i.e., a execução de uma mesma operação, com os mesmos parâmetros, deve produzir o mesmo resultado nas diversas réplicas).

O BFT-SMART [Bessani et al. 2014] é uma biblioteca para implementação de aplicações através de replicação Máquina de Estados [Schneider 1990] que tolera falhas Bizantinas em algumas das réplicas (adicionalmente, o sistema pode ser configurado para tolerar apenas *crashes*). Esta biblioteca *open-source* de replicação foi desenvolvida em Java e implementa um protocolo para RME similar aos outros protocolos para tolerância

a falhas Bizantinas (ex.: [Castro and Liskov 2002]). Além disso, são fornecidos protocolos para reconfiguração e para gerenciamento de estados (*checkpoints*, atualização e transferência de estados).

O BFT-SMART assume um modelo de sistema usual para replicação Máquina de Estados [Castro and Liskov 2002, Bessani et al. 2014]: $n \geq 3f + 1$ servidores para tolerar f falhas Bizantinas; um número ilimitado de clientes que podem falhar; e um sistema parcialmente síncrono para garantir terminação. Estes parâmetros (n e f) podem ser alterados durante a execução através de reconfigurações [Alchieri et al. 2013]. Para comunicação, o sistema ainda necessita de canais ponto-a-ponto autenticados e confiáveis, que são implementados usando MACs (*message authentication codes*) sobre o TCP/IP. As chaves simétricas para a comunicação entre as réplicas são geradas através do protocolo *Signed Diffie-Helman* usando um par de chaves RSA para cada réplica. Já as chaves para a comunicação entre clientes e réplicas são geradas com base nos identificadores dos *endpoints* (cliente e réplica), i.e., não é necessário um par de chaves RSA para cada cliente. Adicionalmente, pode-se configurar os clientes para assinar suas requisições, garantindo-se autenticação das mesmas (neste caso, o par de chaves RSA é necessário).

Resumidamente[1], cada réplica do BFT-SMART realiza as seguintes tarefas:

Recebimento de Requisições. Os clientes enviam suas requisições para as réplicas, que as armazenam em filas diferentes para cada cliente. A autenticidade das requisições é garantida por meio de assinaturas digitais, i.e., os clientes assinam suas requisições (caso configurado). Desta forma, qualquer réplica consegue verificar a autenticidade das requisições e uma proposta para ordenação, que contém a requisição a ser ordenada, somente é aceita por uma réplica correta após a autenticidade desta requisição ser verificada.

Ordenamento de Requisições. Sempre que existirem requisições para serem executadas, um protocolo de difusão atômica é executado onde uma instância de um protocolo de consenso é inicializada por uma réplica (chamada de líder) para definir uma ordem de entrega para um lote de requisições. Caso uma requisição não seja ordenada dentro de um determinado tempo, o sistema troca a réplica líder. Um tempo limite para ordenação é associado a cada requisição r recebida em cada réplica i. Caso este tempo se esgotar, i envia r para todas as réplicas e define um novo tempo para sua ordenação. Isto garante que todas as réplicas recebem r, pois um cliente malicioso pode enviar r apenas para alguma(s) réplica(s), tentando forçar uma troca de líder. Caso este tempo se esgotar novamente, i solicita a troca de líder, que apenas é executada após $f + 1$ réplicas solicitarem esta troca, impedindo que uma réplica maliciosa force trocas de líder.

Execução de Requisições. Quando a ordem de execução de um lote de requisições é definida, este lote é adicionado em uma fila para então ser entregue à aplicação. Após o processamento da cada requisição, uma resposta é enviada ao cliente que solicitou tal requisição. O cliente, por sua vez, determina que uma resposta para sua requisição é válida assim que o mesmo receber pelo mesmo $f + 1$ respostas iguais, garantindo que pelo menos uma réplica correta obteve tal resposta.

[1]Uma descrição mais detalhada do BFT-SMART pode ser encontrada em [Bessani et al. 2014].

3.1. Implementando Aplicações com o BFT-SMART

A forma de utilização do BFT-SMART, para programação de uma aplicação tolerante a falhas através de RME, é bastante simples. O Algoritmo 1 apresenta a API básica para clientes e servidores, mostrando a classe que deve ser instanciada pelos clientes para acessar o sistema, bem como a interface que deve ser estendida pelos servidores para implementar o serviço replicado.

Algoritmo 1 API do BFT-SMART para clientes e servidores.

```
1  //API do Cliente
2  public class ServiceProxy {
3    public ServiceProxy(int id){
4       ...
5    }
6
7    public byte[] invokeOrdered(byte[] request){
8      ...
9    }
10
11   public byte[] invokeUnordered(byte[] request){
12     ...
13   }
14 }
15
16 //API do Servidor
17 public class MyServer extends Executable {
18
19    public MyServer(int id){
20       new ServiceReplica(id, this, ...);
21    }
22
23    public byte[] executeOrdered(byte[] request, MsgContext ctx){
24       //CÓDIGO DA APLICAÇÃO
25    }
26
27    public byte[] executeUnordered(byte[] request, MsgContext ctx){
28       //CÓDIGO DA APLICAÇÃO
29    }
30 }
```

Para acessar o serviço replicado, um cliente do BFT-SMART apenas deve instanciar uma classe *ServiceProxy* fornecendo seu identificador (inteiro) e um arquivo de configuração contendo o endereço (IP e porta) de cada um dos servidores. Após isso, sempre que o cliente desejar enviar alguma requisição para as réplicas (servidores), o mesmo deve invocar o método *invokeOrdered* especificando a requisição (serializada em um *array* de *bytes*). Para requisições que não precisam ordenação (ex.: operações de apenas leitura), o método *invokeUnordered* deve ser utilizado.

Por outro lado, para implementar o servidor, cada réplica deve estender a interface *Executable* e implementar o método abstrato *executeOrdered* que é invocado quando uma requisição deve ser executada. O método *executeUnordered* também deve ser implementado para execução de operações que não precisam ordenação. Além disso, é necessário instanciar uma *ServiceReplica* que representa propriamente a réplica, fornecendo o identificador (inteiro) da réplica que é mapeado para uma porta e endereço IP através de um arquivo de configuração.

Esta é a API básica do BFT-SMART, a qual já é suficiente para implementar uma aplicação tolerante a falhas. No entanto, esta API é muito mais rica, apresentando também

métodos (*getSnapshot* e *installSnapshot*) para o gerenciamento do estado das réplicas (caso deseja-se empregar recuperação de réplicas). Uma descrição mais aprofundada sobre o BFT-SMART pode ser encontrada em [Bessani et al. 2014].

4. Implementando Diversidade em Replicação Máquina de Estados

Esta seção descreve a arquitetura proposta para implementação de diversidade em replicação Máquina de Estados e discute como o mesma foi integrada no BFT-SMART. Em nossa arquitetura, dois problemas principais precisam ser tratados: *(1)* como fazer linguagens diferentes acessarem métodos e funções umas das outras; e *(2)* como trocar informações (dados) entre linguagens diferentes, que também podem adotar diferentes representações de dados.

Para resolver o problema *(1)*, nossa arquitetura utiliza interfaces nativas fornecidas pelas linguagens utilizadas para fazer o programa em Java (linguagem de desenvolvimento do BFT-SMART) executar e/ou receber chamadas de programas em C e, a partir do C, com outras linguagens. Já para resolver o problema *(2)*, nossa arquitetura representa os dados através de *Protocol Buffers* [Protocol Buffers 2016], desenvolvido pela Google em 2008, cujo objetivo é serializar dados de forma universal em qualquer linguagem de programação.

Desta forma, a alternativa utilizada em nossa solução consiste em carregar a JVM (*Java Virtual Machine*) dentro do espaço de memória do programa (que pode ter sido escrito em outra linguagem de programação), realizando chamadas diretas ao ambiente Java. Esta abordagem foi possível pelo fato do Java dar suporte a este tipo de desenvolvimento através da JNI (*Java Native Interface*), que é uma implementação de FFI (*Foreign Function Interface*) para a JVM. Uma FFI possibilita a uma linguagem de alto-nível realizar chamadas para ou a partir de uma linguagem de baixo nível, sem utilização de facilidades IPC (*Inter-Process Communication*) ou RPC (*Remote Procedure Call*).

Potenciais problemas com essa abordagem são conflitos entre a JVM e as outras linguagens de programação, já que ambas operam dentro do mesmo espaço de memória. Um exemplo de problema desse tipo foi encontrado durante o desenvolvimento desta solução, na interface com a linguagem Python: caso ocorresse alguma exceção não-tratada no lado Python, a JVM encerrava abruptamente com um *segfault*. A solução, neste caso, foi criar um tratamento genérico de exceções que, ao ser ativado, encerra corretamente a aplicação.

4.1. Visão Geral da Arquitetura Proposta

Grande parte das linguagens de programação possuem uma FFI para interoperabilidade com a linguagem C, visto que os sistemas operacionais geralmente expõem sua API em linguagem C, e se faz necessário a utilização de chamadas de sistema caso um programa queira utilizar qualquer recurso do sistema operacional, como arquivos ou *sockets*. Por outro lado, poucas linguagens oferecem interoperabilidade direta com a linguagem Java, e em geral, as que fornecem já são linguagens desenvolvidas sobre a JVM.

Dado este cenário, foi concebida uma camada intermediária em C que reside entre o BFT-SMART e as demais linguagens, viabilizando a implementação da diversidade. Efetivamente, foi implementada uma interface de linguagem de alto nível para linguagem

de alto nível, através da introdução da camada intermediária de baixo nível. Por simplicidade, esta camada foi desenvolvida na linguagem C++ e suas funções exportadas para C (utilizando a funcionalidade "*extern* C").

Conforme já comentado, ainda existe a necessidade de transferência de dados entre diferentes linguagens de programação. Como cada linguagem possui uma forma distinta de representação dos dados (*e.g.* objetos em Java *versus* structs em C), faz-se necessário o desenvolvimento de um protocolo em comum para troca de informações entre as diferentes linguagens de programação. Para tal propósito foi utilizada a biblioteca *Protocol Buffers* [Protocol Buffers 2016]. Esta biblioteca trabalha com uma linguagem própria de descrição de dados: arquivos `.proto` que são compilados pelo *Protocol Buffers*, gerando código em múltiplas linguagens de programação, com funções/métodos para codificação e decodificação de dados em *arrays* de *bytes*.

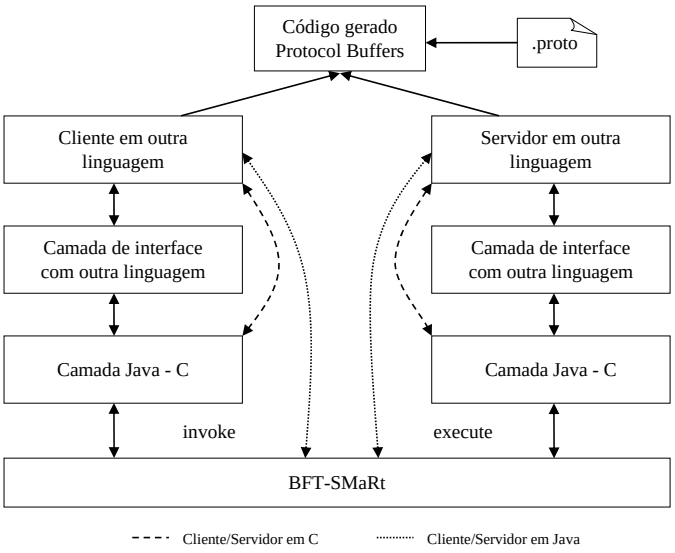


Figura 1. Camadas de interoperabilidade.

A Figura 1 apresenta a arquitetura em camadas da solução proposta. A ideia básica é que clientes escritos em qualquer linguagem utilizem o *Protocol Buffers* para transformar os dados que desejam transmitir (na requisição) em um *array* de *bytes*. Então, é realizada uma comunicação com a linguagem C (camadas intermediárias) que comunica-se tanto com a linguagem na qual foi escrito o cliente quanto com o Java (linguagem em que o BFT-SMART foi escrito). Como as interfaces do BFT-SMART para requisições (e respostas) são definidas através de *arrays* de *bytes* (veja Algoritmo 1), os dados inicialmente transformados em *bytes* pelo cliente através do *Protocol Buffers* são transmitidos até o BFT-SMART que emprega seus protocolos de RME para fazer com que os mesmos sejam entregues para serem executados na mesma ordem pelos servidores. Para isso, é realizado o caminho oposto do descrito anteriormente, i.e., os dados passam do BFT-SMART para as camadas seguintes até chegar na última camada (servidores).

Cada servidor então utiliza o *Protocol Buffers* para reconstituir os dados na sua linguagem de implementação, executa a devida requisição e obtém a resposta. Após isso, para enviar a resposta ao cliente, é realizado o mesmo processamento descrito anteriormente para a requisição. A única diferença é que neste caso o BFT-SMART aguarda por

uma maioria de respostas iguais para então considerar a requisição executada com sucesso e repassar a resposta para as camadas seguintes até chegar ao cliente.

Nesta arquitetura, uma garantia fundamental fornecida pelo *Protocol Buffers*, que possibilita esta integração com o BFT-SMART, é que se os dados são equivalentes então a representação binária dos mesmos é idêntica. Isso possibilita ao BFT-SMART trabalhar somente com os *arrays* de *bytes* (o código em Java do BFT-SMART não possui nenhum conhecimento sobre as estruturas de dados que está transmitindo) e mesmo assim consegue comparar as requisições/respostas e o estado das réplicas para realizar as contagens dos quóruns necessários nos protocolos de RME [Castro and Liskov 2002, Bessani et al. 2014] a fim de manter as propriedades do sistema.

Vale destacar que para clientes ou servidores escritos em Java, as camadas intermediárias não são necessárias. Já para clientes ou servidores escritos em C ou C++, as duas últimas camadas são aglutinadas, i.e., não é necessário utilizar uma FFI para comunicação entre C e estas linguagens.

4.2. Interface com o BFT-SMART

Esta seção descreve em maiores detalhes a interface escrita na linguagem de programação C para comunicação com o BFT-SMART. Em geral, FFIs são compostas por três funcionalidades principais:

a. Utilização de funções escritas em C;
b. Utilização, a partir do C, de funções escritas em outras linguagens; e
c. Conversão de tipos entre as linguagens: entre parâmetros e valores de retorno.

O Algoritmo 2 apresenta a interface em C para comunicação com o BFT-SMART. Tanto clientes quanto servidores, ao iniciar, devem carregar a JVM, opcionalmente configurando um *classpath* customizado, através das funções *setClasspath* e *carregarJvm*.

Algoritmo 2 API do BFT-SMART para clientes e servidores em C.

```
1   void setClasspath(const char* cp);
2   int carregarJvm();
3   void finalizarJvm();
4   int createServiceProxy(int id);
5   int startServiceReplica(int id);
6   int invokeOrdered(BFT_BYTE command[], int tamanho, BFT_BYTE saida[]);
7   int invokeUnordered(BFT_BYTE command[], int tamanho, BFT_BYTE saida[]);
8   void implementExecuteOrdered(int (*impl) (BFT_BYTE [], int, BFT_BYTE **));
9   void implementExecuteUnordered(int (*impl) (BFT_BYTE [], int, BFT_BYTE **));
10  void implementInstallSnapshot(void (*impl) (BFT_BYTE [], int));
11  void implementgetSnapshot(int (*impl) (BFT_BYTE **));
12  void implementReleaseExecuteOrderedBuffer(void (*impl) (BFT_BYTE*));
13  void implementReleaseExecuteUnorderedBuffer(void (*impl) (BFT_BYTE*));
14  void implementReleaseGetSnapshotBuffer(void (*impl) (BFT_BYTE*));
15  void * bftsmartallocate(size_t tamanho);
16  void bftsmartrelease(void * ponteiro) ;
```

Após carregar a JVM, clientes devem invocar a função *createServiceProxy* para criar o *ServiceProxy* utilizado para acessar a RME enquanto que os servidores devem invocar a função *startServiceReplica* para iniciar suas operações (Seção 3.1). A partir deste ponto, os clientes podem realizar requisições através das funções *invokeOrdered* e *invokeUnordered*.

No caso dos servidores, funções escritas em C serão invocadas pelo BFT-SMART. Nesse caso, a assinatura das funções são descritas como métodos em uma classe Java, com a palavra-chave "`native`". Em tempo de execução, cada um desses métodos é associado com um ponteiro para uma função em C. Quando o Java invoca esses métodos, a função correspondente em C é chamada. Sendo assim, servidores devem fornecer uma implementação para as funções *executeOrdered*, *executeUnordered*, *installSnapshot* e *getSnapshot* (estas duas últimas para gerenciamento de estado [Bessani et al. 2014]). Isto é realizado através das funções *implement**, que recebem ponteiros para função.

Além disso, devem ser fornecidas implementações para as funções *implementRelease**. Como não há garantia do tamanho da resposta que vem do servidor, é realizada uma alocação dinâmica de memória, na qual a resposta é armazenada. Após o processamento da resposta pelo ambiente Java, o BFT-SMART chama essas funções para sinalizar que a memória já pode ser liberada.

4.3. Clientes e Servidores com Diversidade

Atualmente, é possível implementar clientes e servidores diversificados em quatro linguagens: Java, C, C++ e Python. A implementação em Java é direta uma vez que o BFT-SMART é escrito em Java, bastando usar o *Protocol Buffers* para codificar os dados e, com isso, possibilitar a troca de dados com réplicas e clientes implementados em outras linguagens. Já para implementações em C e C++, as duas últimas camadas de nossa arquitetura (Figura 1) são aglutinadas, pois não há necessidade de utilização de uma FFI (a comunicação é direta entre a "Camada Java – C" e C ou C++).

Finalmente, a implementação em Python é realizada através da interface apresentada no Algoritmo 2, sendo que foi criada uma interface para realizar as chamadas a esta camada intermediária em C. Deste modo, para implementar clientes e servidores em Python, basta herdar de uma classe abstrata para implementar as funções de forma idiomática, e essa última camada cuida de realizar as traduções devidas para C. Para incluir suporte a outras linguagens de programação, basta utilizar a camada intermediária em C (Algoritmo 2) para comunicação com a nova linguagem de programação.

Vale destacar que os servidores, mesmo implementados em linguagens diferentes, devem apresentar determinismo na execução das operações. A ferramenta DiveInto [Antunes and Neves 2011] possibilita analisar a conformidade entre diferentes implementações de servidores e encontrar inconsistências.

5. Experimentos

Visando analisar o desempenho da arquitetura proposta e da implementação desenvolvida (Seção 5.1), bem como discutir aspectos relacionados com a segurança de aplicações replicadas por meio de uma RME com diversidade (Seção 5.2), alguns experimentos foram realizados no Emulab [White et al. 2002]. O BFT-SMART foi configurado com $n = 4$ servidores para tolerar até 1 falha Bizantina. Cada servidor executou em uma máquina separada, enquanto que 100 clientes executaram em outra máquina. Todos os experimentos realizados tiveram uma fase inicial de *warm-up*.

Aplicação. Uma aplicação de lista, implementada nos servidores como lista encadeada, foi desenvolvida em várias linguagem (Java, C, C++ e Python). Nestes experimentos, a lista foi utilizada para armazenar inteiros e as seguintes operações foram implementadas

para seu acesso: `ADD` – que adiciona um inteiro no final da lista e retorna 1 caso este inteiro ainda não esteja na lista, retorna 0 caso contrário; `REMOVE` – que remove um inteiro passado como parâmetro caso esteja na lista; retorna um código de erro caso contrário; e `GET` – que retorna o elemento da posição passada como parâmetro ou um código de erro caso não exista um elemento na posição indicada (maior que o tamanho da lista).

Em nossos experimentos, a lista foi inicializada com 200k entradas em cada réplica e os valores para as operações `ADD`, `GET` e `REMOVE` foram selecionados aleatóriamente seguindo uma distribuição uniforme. Além disso, todos os clientes executados foram desenvolvidos na linguagem Java, enquanto que os servidores foram diversificados conforme os cenários descritos em cada experimento.

A latência e o *throughput* destas operações foram determinados, visando analisar o desempenho do sistema nos diversos cenários. A latência foi medida em um dos clientes e os valores apresentados representam a média de 1000 execuções, excluindo-se 10% dos valores com maior desvio. Já o *throughput* apresentado é o pico atingido pelos servidores, medido no servidor líder do consenso [Bessani et al. 2014] a cada 5000 requisições.

5.1. Análise do Desempenho: Sem Diversidade no Ambiente de Execução

Primeiramente, executamos alguns experimentos em um ambiente onde todas as máquinas possuiam a mesma configuração, o que possibilitou uma análise mais precisa a respeito do *overhead* introduzido pelo uso de diversidade. Desta forma, o ambiente consistiu de 5 máquinas *d710* (2.4 GHz 64-*bit* Intel Quad Core Xeon E5530, 12GB de RAM e interface de rede gigabit) conectadas a um *switch* de 1Gbps. O ambiente de *software* utilizado foi o sistema operacional Ubuntu 12.04 com *kernel* 3.2.0, JVM Oracle JDK 1.7.0_79 (Java), g++ 4.6.0 (C++), gcc 4.8.3 (C) e Python 2.7.3 (Python).

Resultados e Análises. A Tabela 1 apresenta os valores para o *throughput* e a latência em três cenários distintos: *(1)* apenas usando Java, i.e., o BFT-SMART sem diversidade; *(2)* usando Java e *Protocol Buffers* (Java + PB), que possibilita a avaliação do *overhead* introduzido pelo mecanismo de representação dos dados; e *(3)* usando cada réplica do BFT-SMART em uma linguagem diferente (Java, C, C++ e Python), que possibilita avaliar toda a arquitetura proposta, incluindo o *overhead* tanto do mecanismo de representação dos dados quanto para uma linguagem executar métodos ou funções de outras linguagens.

		Java	Java + PB	Java, C, C++ e Python	Java, C, C++ e C++
`ADD`	Throughput (kop/seg)	11.55	11.32	9.25	10.59
	Latência (ms)	34.92	35.56	100.43	47.46
	Desvio Padrão (ms)	10.10	9.22	46.15	15.48
`GET`	Throughput (kop/seg)	4.45	4.40	0.85	2.08
	Latência (ms)	24.33	24.08	126.63	55.61
	Desvio Padrão (ms)	2.06	2.22	7.73	5.97
`REMOVE`	Throughput (kop/seg)	2.66	2.55	0.48	1.22
	Latência (ms)	47.06	47.91	231.23	90.44
	Desvio Padrão (ms)	5.01	4.19	9.99	7.19

Tabela 1. Experimentos sem diversidade no ambiente de execução.

Podemos perceber que a variação no desempenho introduzida pela utilização de *Protocol Buffers* é praticamente desprezível. Porém, o *throughput* diminui quando o sistema é configurado com uma réplica em cada linguagem, comportamento que fica mais

evidente para as operações `GET` e `REMOVE`. A latência também se comportou de forma semelhante, aumentando quando réplicas em diferentes linguagens foram utilizadas.

Visando analisar os fatores que levaram a esta queda de desempenho, medimos o tempo que cada operação leva para ser executada em cada implementação (execução da operação $executeOrdered$ na linguagem específica para as operação `ADD`, `GET` e `REMOVE`). Como podemos observar na Tabela 2, a implementação em Python apresentou um tempo de resposta muito superior as demais, enquanto C e C++ tiveram desempenhos próximos mas que são praticamente o dobro do que em Java. Vale destacar que estes valores não sofrem influência da arquitetura para diversidade proposta neste trabalho, pois estes valores foram medidos já na linguagem específica (Java, C, C++ e Python).

Como a réplica em Python ficava muito atrasada em relação as demais, a mesma iniciava os protocolos para transferência e atualização de estados [Bessani et al. 2014]. Durante este procedimento, esta réplica solicitava o estado que era transferido a partir das outras réplicas, o que acaba impactando no desempenho do sistema. Para evidenciar este comportamento, na última coluna da Tabela 1 são apresentados os resultados para estes experimentos trocando a réplica em Python por outra réplica em C++. Comparando com a utilização de apenas Java, podemos perceber que neste caso o desempenho melhora, ficando muito próximo para a operação de `ADD` e praticamente a metade para as outras operações, o que é explicado pelo fato de as operações demorarem praticamente o dobro em C++ (Tabela 2). O *throughput* da operação `ADD` fica muito próximo da configuração com apenas Java porque os resultados apresentados referem-se ao pico atingido pelo sistema e, como inicialmente a lista está vazia, este valor é conseguido logo no começo (a queda no desempenho em Python está relacionada com as buscas que ocorrem na lista).

		Java	C	C++	Python
`ADD`	Tempo de execução (ms)	0.33	0.71	0.55	11.45
	Desvio Padrão (ms)	0.09	0.17	0.13	2.45
`GET`	Tempo de execução (ms)	0.20	0.70	0.43	7.68
	Desvio Padrão (ms)	0.10	0.32	0.10	3.36
`REMOVE`	Tempo de execução (ms)	0.51	1.09	0.67	14.03
	Desvio Padrão (ms)	0.06	0.19	0.12	0.51

Tabela 2. Tempo de resposta das linguagens (execução de $executeOrdered$).

Para certificar-se de que a queda de desempenho está relacionada com o processamento da requisição em uma determinada linguagem, analisamos uma aplicação vazia (nada é processado nos servidores, que apenas retornam uma resposta). Este tipo de aplicação é comumente utilizado para avaliar estes sistemas [Bessani et al. 2014] e apenas os tamanhos das requisições/respostas são configurados. A Tabela 3 apresenta os resultados para a configuração com requisições/respostas de tamanho $0/0$. Podemos perceber que neste caso o desempenho é praticamente o mesmo em todos os cenários.

	Java	Java + PB	Java, C, C++ e Python
Throughput (kops/seg)	48.55	48.35	47.17
Latência (ms)	2.24	2.28	2.29
Desvio Padrão (ms)	0.23	0.29	0.36

Tabela 3. Experimento para um aplicação vazia ($0/0$).

5.2. Análise da Segurança: Com Diversidade no Ambiente de Execução

Apesar da aplicação apresentar diversidade de implementação nos experimentos anteriores, a mesma configuração de execução foi utilizada em cada servidor de forma que uma mesma vulnerabilidade pode comprometer toda a aplicação. Por exemplo, uma vulnerabilidade no sistema operacional utilizado (Ubuntu) pode ser explorada em todas as réplicas, visto que todas executam sobre este mesmo sistema operacional. Neste sentido, esta seção apresenta experimentos executados em um ambiente que contempla outros eixos de diversidade, o que aumenta a segurança das aplicações conforme discutido a seguir.

Configuração do Ambiente de Execução e Análise da Segurança. Visando aumentar a segurança através de diversidade, os ambientes de execução das réplicas devem ser diferentes, de forma que seja pouco provável que uma mesma vulnerabilidade esteja presente em mais de um deles. A Tabela 4 apresenta a configuração adotada em cada réplica, que novamente foram conectadas a um *switch* de 1Gbps. A nomenclatura utilizada para o *hardware* é aquela fornecida pelo Emulab [White et al. 2002]. Dentre as configurações disponibilizadas pela plataforma, utilizamos uma configuração diferente para cada servidor. Também utilizamos diferentes distribuições e/ou versões de sistemas operacionais, além de compiladores e/ou ambientes de execução para as linguagens utilizadas. O C++ está presente em todas as réplicas (menos em Java) visto que a camada intermediária foi desenvolvida em C++ e teve sua interface exportada para C (Seção 4).

Eixo de Diversidade		Réplica 1	Réplica 2	Réplica 3	Réplica 4
Aplicação		Java	C++	C	Python
Sistema Operacional		Fedora Core 15 *kernel* 2.6.40	FreeBSD 10.0	CentOS 7.1 *kernel* 3.10.0	Ubuntu 12.04 *kernel* 3.2.0
COTS	JVM (Java)	Oracle JDK 1.7.0_79	OpenJDK 1.8.0_60	Oracle JDK 1.8.0_60	OpenJDK 1.7.0_91
	C	–	–	gcc 4.8.3	–
	C++	–	clang 3.3	g++ 4.8.3	g++ 4.6.3
	Python	–	–	–	Python 2.7.3
Hardware		pc2400w	pc3000	d820	d710

Tabela 4. Configuração das réplicas com diversidade.

Como podemos perceber, um atacante não é capaz de explorar uma mesma vulnerabilidade para comprometer mais de uma réplica. Por exemplo, uma vulnerabilidade no sistema operacional Ubuntu comprometeria apenas a réplica 4, enquanto que uma falha no *hardware* pc3000 afetaria apenas a réplica 2, e assim por diante. Desta forma, a segurança da aplicação é aumentada na medida em que mais vulnerabilidades são necessárias para comprometer o sistema. De fato, é necessário o comprometimento de mais de uma réplica para que o sistema deixe de funcionar corretamente.

Resultados e Análises. A Tabela 5 apresenta os resultados para o sistema com diversidade nos ambientes de execução. Considerando aspectos de desempenho, o comportamento é semelhante aos experimentos anteriores, apresentando uma queda nos cenários com uso de diversidade. No entanto, quando consideramos aspectos de segurança, o sistema fica mais robusto visto que uma combinação de vulnerabilidades (e não apenas uma) é necessária ser explorada por um atacante para comprometer o sistema.

		Java	Java + PB	Java, C, C++ e Python
ADD	Throughput (kop/seg)	4.97	4.62	1.16
	Latência (ms)	27.32	29.69	131.74
	Desvio Padrão (ms)	6.52	8.89	41.70
GET	Throughput (kop/seg)	10.75	10.82	1.54
	Latência (ms)	9.75	9.55	71.02
	Desvio Padrão (ms)	1.10	0.97	8.40
REMOVE	Throughput (kop/seg)	6.61	6.29	1.05
	Latência (ms)	16.18	16.72	106.63
	Desvio Padrão (ms)	1.68	1.176	13.28

Tabela 5. Experimentos com diversidade no ambiente de execução.

6. Conclusões e Trabalhos Futuros

Este artigo apresentou uma nova abordagem para RME que utiliza diversidade para aumentar a segurança das aplicações. De fato, em um sistema diversificado, é muito pouco provável que uma mesma vulnerabilidade possa ser explorada em mais de uma réplica. A arquitetura proposta para suporte à diversidade foi implementada no BFT-SMART e através de uma série de experimentos verificou-se o comportamento na prática de uma RME com diversidade. De um modo geral, o desempenho do sistema como um todo fica condicionado ao pior desempenho dentre as implementações (linguagens) utilizadas. Porém, a segurança do sistema é aumentada, fator primordial para muitas aplicações. As implementações desenvolvidas estão disponíveis no seguinte repositório: `https://github.com/caioycosta/bftsmart-diversity`.

Como trabalhos futuros, pretende-se implementar suporte para outras linguagens de programação, além de explorar outras aplicações a fim de aumentar a compreensão sobre o funcionamento de uma RME com diversidade. Também pretendemos analisar os efeitos da utilização de diversidade nos clientes, embora este não seja o objetivo principal.

Referências

Alchieri, E. A. P., Bessani, A. N., and da Silva Fraga, J. (2013). Replicação máquina de estados dinâmica. In *Anais do XIV Workshop de Teste e Tolerância a Falhas*.

Amir, Y., Coan, B., Kirsch, J., and Lane, J. (2011). Prime: Byzantine replication under attack. *IEEE Transactions on Dependable and Secure Computing*, 8(4):564–577.

Antunes, J. and Neves, N. (2011). DiveInto: Supporting diversity in intrusion-tolerant systems. In *30th IEEE Symposium on Reliable Distributed Systems*.

Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33.

Avizienis, A. and Chen, L. (1977). On the implementation of n-version programming for software fault tolerance during execution. In *International Computer Software and Applications Conference*.

Bessani, A., Daidone, A., Gashi, I., Obelheiro, R., Sousa, P., and Stankovic, V. (2009). Enhancing fault / intrusion tolerance through design and configuration diversity. In *Proceedings of the 3rd Workshop on Recent Advances on Intrusion-Tolerant Systems*.

Bessani, A., Sousa, J., and Alchieri, E. (2014). State machine replication for the masses with BFT-SMaRt. In *Proceedings of the International Conference on Dependable Systems and Networks*.

Castro, M. and Liskov, B. (2002). Practical Byzantine fault-tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461.

Castro, M., Rodrigues, R., and Liskov, B. (2003). BASE: Using abstraction to improve fault tolerance. *ACM Transactions Computer Systems*, 21(3):236–269.

Clement, A., Kapritsos, M., Lee, S., Wang, Y., Alvisi, L., Dahlin, M., and Riché, T. (2009a). UpRight cluster services. In *Proceedings of the ACM Symposium on Operating Systems Principles*.

Clement, A., Wong, E., Alvisi, L., Dahlin, M., and Marchetti, M. (2009b). Making byzantine fault tolerant systems tolerate byzantine faults. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*.

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

Garcia, M., Bessani, A., Gashi, I., Neves, N., and Obelheiro, R. (2011). Os diversity for intrusion tolerance: Myth or reality? In *Proceedings of the IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, DSN '11.

Garcia, M., Bessani, A., Gashi, I., Neves, N., and Obelheiro, R. (2014). Analysis of operating system diversity for intrusion tolerance. *Software: Practice and Experience*, 44(6):735–770.

Guerraoui, R., Knežević, N., Quéma, V., and Vukolić, M. (2010). The next 700 BFT protocols. In *Proceedings of the ACM SIGOPS/EuroSys European Systems Conference*.

Hadzilacos, V. and Toueg, S. (1994). A modular approach to the specification and implementation of fault-tolerant broadcasts. Technical report, Department of Computer Science, Cornell.

Kotla, R., Alvisi, L., Dahlin, M., Clement, A., and Wong, E. (2009). Zyzzyva: Speculative Byzantine fault tolerance. *ACM Transactions on Computer Systems*, 27(4):7:1–7:39.

Obelheiro, R. R., Bessani, A. N., and Lung, L. C. (2005). Analisando a viabilidade da implementação prática de sistemas tolerantes a intrusões. In *Anais do V Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais - SBSeg 2005*.

Platania, M., Obenshain, D., Tantillo, T., Sharma, R., and Amir, Y. (2014). Towards a practical survivable intrusion tolerant replication system. In *33rd IEEE International Symposium on Reliable Distributed Systems*, pages 242–252.

Protocol Buffers (2016). Protocol buffers developers. Disponível em *https://developers.google.com/protocol-buffers/*. Último acesso em Abril de 2016.

Randell, B. (1975). System structure for software fault tolerance. In *Proceedings of the International Conference on Reliable Software*.

Schneider, F. B. (1990). Implementing fault-tolerant service using the state machine aproach: A tutorial. *ACM Computing Surveys*, 22(4):299–319.

Veronese, G., Correia, M., Bessani, A., Lung, L., and Verissimo, P. (2013). Efficient Byzantine fault tolerance. *IEEE Transactions on Computers*, 62(1).

Veríssimo, P., Neves, N. F., and Correia, M. P. (2003). Intrusion-tolerant architectures: Concepts and design. In *Architecting Dependable Systems*, volume 2677 of *Lecture Notes in Computer Science*. Springer-Verlag.

White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. (2002). An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. of 5th Symp. on Operating Systems Design and Implementations*.

Uma Implementação MPI Tolerante a Falhas do Algoritmo Hyperquicksort

Edson Tavares de Camargo[1,2], **Elias P. Duarte Jr.**[2]

[1] Universidade Federal do Paraná (UFPR) – Programa de Pós-Graduação em Informática
Caixa Postal 19081 – 81531-980 – Curitiba – PR – Brasil

[2]Universidade Tecnológica Federal do Parará - Campus Toledo (UTFPR)
CEP: 85902-490 – Toledo – PR – Brasil

edson@utfpr.edu.br, elias@inf.ufpr.br

Abstract. *Hyperquicksort is a parallel sorting algorithm based on a virtual hypercube. Current implementations of this algorithm do not support fault tolerance. This paper presents an implementation of Hyperquicksort that is able to tolerate up to* $n - 1$ *faulty processes, where n is the total number of processes employed for sorting. Our fault-tolerant Hyperquicksort version was implemented in accordance to the latest MPI fault tolerance specification, the User Level Failure Mitigation (ULFM). A implementation is described in which Hyperquicksort manages to reconfigure itself at runtime proceeding with its execution despite faulty processes. Experimental results show the efficiency of the implementation in ordering up to 1 billion integers.*

Resumo. *O Hyperquicksort é um algoritmo de ordenação paralela baseado em um hipercubo virtual. As implementações disponíveis desse algoritmo não consideraram a tolerância a falhas. Este trabalho apresenta uma implementação do Hyperquicksort que é capaz de tolerar até n-1 falhas de processos, onde n é o número total de processos empregados na ordenação. O Hyperquicksort tolerante a falhas foi implementado de acordo com a mais recente especificação de tolerância a falhas do MPI, a User Level Failure Mitigation (ULFM). Dessa forma, o Hyperquicksort é programado para se reconfigurar e continuar a sua execução, apesar de falhas de processos. Resultados mostram a eficiência da implementação na ordenação de até 1 bilhão de números inteiros.*

1. Introdução

A ordenação é uma das operações fundamentais em computação [Chen and Chung 2001]. O *Hyperquicksort* [Wagar 1987] é a versão paralela do algoritmo sequencial *Quicksort* [Hoare 1962] baseado na topologia de um hipercubo virtual. O hipercubo é uma estrutura largamente utilizada como topologia de interligação e comunicação e para a execução de algoritmos paralelos e distribuídos [Parhami 1999, Duarte, Jr. and Nanya 1998]. As propriedades do hipercubo como, por exemplo, diâmetro logarítmico, além do fato de ser uma estrutura recursiva altamente simétrica, o fazem suportar uma variedade de algoritmos paralelos elegantes e eficientes [Leighton 1991].

Basicamente, o *Hyperquicksort* realiza a ordenação da seguinte forma. Cada processo recebe uma lista de números de igual tamanho. A ordenação ocorre em rodadas de ordenação. A cada rodada pares de processos são formados e trocam partes da

sua lista de números entre si de acordo com um número pivô. Ao final de $log_2 n$ rodadas de ordenação (onde n é uma potência de 2 e representa o número total de processos) cada processo tem sua lista ordenada localmente. O maior número contido na lista do processo com menor identificador é menor ou igual ao menor número contido na lista do processo com maior identificador. Além do *Hyperquicksort*, existem outros algoritmos de ordenação baseados no hipercubo [Johnsson 1984, Won and Sahni 1988, Seidel and George 1988, Plaxton 1989]. No entanto, nenhum desses fornece a capacidade de tolerar falhas [Chen and Chung 2001]. Embora os algoritmos definidos nos trabalhos [Chen and Chung 2001, Sheu et al. 1992], tolerem algumas falhas de nodos os mesmos são propostos no âmbito do *BitonicSort* [Johnsson 1984].

O MPI (*Message-Passing Interface*) é o padrão de *facto* para o desenvolvimento de aplicações paralelas e distribuídas. O MPI baseia-se no paradigma de troca de mensagens, onde os processos, ou nodos, acessam uma memória local e estão conectados através de uma rede. O padrão assume que a infraestrutura subjacente é confiável [MPI Forum 2015]. Dessa forma, o MPI não define o comportamento que as implementações devem adotar perante falhas [Bland et al. 2013, Gropp and Lusk 2004]. Consequentemente, as implementações MPI amplamente usadas, como a OpenMPI [Gabriel et al. 2004] e a MPICH [Gropp et al. 1996], abortam toda a aplicação perante a falha de um único processo.

Este trabalho apresenta uma implementação MPI tolerante a falhas do algoritmo paralelo de ordenação *Hyperquicksort*. A implementação é capaz de tolerar até $n - 1$ falhas de processos em tempo de execução. Na implementação, foi utilizada a mais recente especificação de tolerância a falhas em MPI, a *User Level Failure Mitigation* (ULFM) [Bland et al. 2012a] proposta pelo MPI-Fórum. A ULFM oferece um conjunto mínimo de interfaces para recuperar a capacidade do MPI de continuar transportando suas mensagens após uma falha. Uma estratégia de comunicação global das falhas é implementada através da ULFM. A partir de então, uma função de mapeamento é aplicada para permitir que os processos que não falharam assumam as funções dos processos falhos e continuem a computação. Ao final do processo de ordenação as listas inicialmente distribuídas a cada processo estarão ordenadas conforme prevê o *Hyperquicksort*. Resultados experimentais são apresentados para a ordenação de até 1 bilhão de inteiros e confirmam a eficiência da implementação tolerante a falhas.

Este trabalho segue organizado da seguinte forma. A Seção 2 apresenta o modelo de sistema e as definições básicas. A Seção 3 apresenta o padrão MPI e a proposta de tolerância a falhas do MPI-Fórum. O algoritmo *Hyperquicksort* e a sua versão tolerante a falhas é apresentada na Seção 4. A implementação é apresentada na Seção 5. Os resultados experimentais na Seção 6. Por fim, a conclusão é apresentada na Seção 7.

2. Modelo de Sistema e Definições

Os processos se comunicam por operações de envio e recebimento de mensagens através de uma rede. A rede é representada por um grafo completo. Os processos estão organizados em uma topologia de hipercubo virtual. Um hipercubo de d dimensões possui 2^d processos. A Figura 1 apresenta um hipercubo de 3 dimensões. Cada processo i é identificado pelo código binário $(n_0, ..., n_{2^s-1})$ do seu identificador. Dois processos estão conectados se seus endereços diferem em apenas um bit, conforme ilustrado na Figura 1. A comunicação é confiável, garantindo que mensagens trocadas entre dois processos não

são perdidas, corrompidas ou duplicadas. Falhas de particionamento não são tratadas.

O modelo de falhas é o *fail-stop*: um processo falha permanentemente e os demais processos podem detectar a falha [Freiling et al. 2011]. Um processo possui um entre dois estados possíveis: falho ou sem-falha. Um processo que nunca falha é considerado correto ou sem-falha. Falhas são detectadas por um serviço de detecção de falhas perfeito, isto é, nenhum processo é detectado como falho sem estar falho e todo processo falho é detectado por todos os processos corretos em um tempo finito [Chandra and Toueg 1996]. Os processos têm acesso a um diretório compartilhado que é confiável, ou seja, não falha.

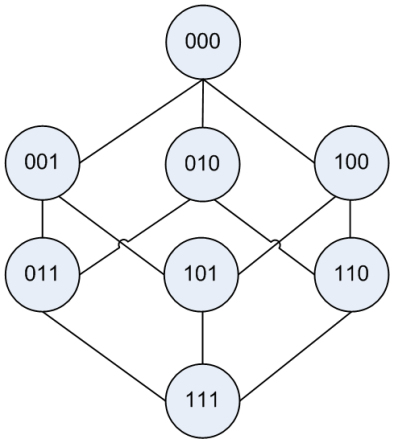


Figura 1. Hipercubo de 3 dimensões.

3. O Padrão MPI

O padrão MPI (*Message Passing Interface*) oferece um dos principais modelos para o desenvolvimento de aplicações paralelas e distribuídas baseado no paradigma de troca de mensagens. O paradigma de troca de mensagens se destina a ambientes computacionais em que os nodos acessam uma memória local e estão conectados através de uma rede - que pode ser tanto um barramento de alta velocidade quanto uma rede local de computadores. Embora o MPI seja baseado no paradigma de troca de mensagens, o padrão também pode ser utilizado em computadores que fazem uso de memória compartilhada.

O MPI consiste de um conjunto de bibliotecas de funções padronizadas pelo MPI-Fórum[1]. Há rotinas para comunicação direta entre dois processos, chamada de comunicação ponto-a-ponto, e rotinas para comunicação coletiva. Além disso, há primitivas para o gerenciamento e criação de processos, entrada e saída de dados em paralelo, gerenciamento de grupos e sincronização de processos e o estabelecimento de topologias virtuais. O MPI-Fórum é a entidade composta por pesquisadores, desenvolvedores e organizações responsáveis por desenvolver e manter a norma MPI, atualmente na sua versão 3.1 [MPI Forum 2015]. Entre as principais implementações da norma MPI se destacam a MPICH [Gropp et al. 1996] e a Open MPI [Gabriel et al. 2004].

Um conceito fundamental em MPI é o comunicador (*communicator*), uma importante estrutura de dados que define o contexto da comunicação e o conjunto de processos pertencentes a esse contexto. No comunicador, os processos são identificados

[1]http://www.mpi-forum.org/

unicamente por meio de um número inteiro positivo chamado *rank*. Há um comunicador pré-definido chamado `MPI_COMM_WORLD` que reúne todos os processos disponíveis no início da execução de uma aplicação ou programa MPI. Um programa MPI pode possuir um ou mais comunicadores.

Uma propriedade fundamental, porém ausente na especificação MPI, é a tolerância a falhas [Gropp and Lusk 2004]. O padrão MPI assume que a infraestrutura subjacente é totalmente confiável [MPI Forum 2015]. Dessa forma, o padrão não define o comportamento preciso que as implementações MPI devem adotar perante falhas [Bland et al. 2013, Gropp and Lusk 2004]. Basicamente, uma falha é tratada como um erro interno da aplicação como, por exemplo, a violação de um espaço de memória. Dessa forma, as falhas de processo ou de rede são repassadas à aplicação simplesmente como se fossem erros de chamadas de funções. Consequentemente, desloca-se a responsabilidade de detectar e de tratar as falhas para as implementações MPI. A norma define os manipuladores de erros (*error handlers*), que são associados ao comunicador MPI, para lidar com os erros do programa.

O manipulador de erros `MPI_ERRORS_ARE_FATAL` é associado por padrão ao comunicador `MPI_COMM_WORLD` e especifica que a manifestação de um erro durante a chamada de uma função MPI leva a todos processos no comunicador a abortar sua execução, encerrando assim toda a aplicação. Por outro lado, o manipulador `MPI_ERRORS_RETURN` retorna um código de erro que indica que uma falha ocorreu [MPI Forum 2015]. Mesmo que um código de erro seja retornado ao programa MPI, o padrão MPI não estabelece mecanismos para lidar com as falhas. Por essa razão, o suporte a tolerância a falhas pela norma MPI é considerado inadequado [Bland et al. 2012b, Bland et al. 2012c]. Além disso, as duas principais implementações MPI citadas acima adotam o manipulador de erro `MPI_ERRORS_ARE_FATAL` por padrão e não dão suporte adequado ao manipulador de erros `MPI_ERRORS_RETURN`, impedindo a continuidade da aplicação no caso de falha.

Diversos trabalhos visam adicionar à implementação MPI rotinas específicas para lidar com as falhas. Entre esses estão o FT-MPI [Fagg and Dongarra 2000], FT/MPI [Batchu et al. 2004], Gropp e Lusk [Gropp and Lusk 2004] e o NR-MPI [Suo et al. 2013]. No entanto, os mesmos não foram adotados pela norma MPI e foram descontinuados. De forma a padronizar e incorporar a tolerância a falhas na norma MPI, o MPI-Fórum criou um grupo de trabalho específico. Os esforços do grupo de trabalho resultaram em duas propostas: a RTS[2] (*Run-through Stabilization Proposal*) [Hursey et al. 2011] e a ULFM[3] (*User-Level Failure Mitigation*) [Bland et al. 2012a, Bland et al. 2013]. A RTS foi a primeira proposta do grupo de tolerância a falhas. Devido à complexidade presente na implementação das primitivas, a proposta RTS não prosseguiu o seu desenvolvimento. A seguir a especificação ULFM é apresentada.

3.1. A Especificação ULFM

A especificação ULFM é o mais recente esforço do MPI-Fórum para padronizar a semântica de tolerância a falhas em MPI[4]. A implementação da ULFM está em desenvolvimento

[2]Disponível em https://svn.mpi-forum.org/trac/mpi-forum-web/wiki/ft/run_through_stabilization

[3]Disponível em http://fault- tolerance.org/ulfm/ulfm-specification/

[4]Disponível em http://meetings.mpi-forum.org/MPI_4.0_main_page.php

como um subprojeto do projeto Open MPI[5] [Gabriel et al. 2004]. Existe a expectativa de que a adotação da ULFM pelo padrão MPI se dê a partir das próximas versões da norma MPI[6].

A ULFM oferece um conjunto mínimo de interfaces para recuperar a capacidade do MPI de continuar transportando suas mensagens após uma falha. Não há uma estratégia de recuperação específica. O objetivo é permitir que o desenvolvedor escolha a técnica de tolerância a falhas que melhor se adeque ao programa. A compatibilidade de código com as versões anteriores do MPI também está entre os requisitos observados.

A aplicação é notificada da falha de um processo ao tentar se comunicar diretamente (comunicação ponto-a-ponto, por exemplo através de um `MPI_Send()` e um `MPI_Recv`) ou indiretamente (por exemplo através de um `MPI_Bcast()`) com o processo falho. A ULFM adota o modelo de falhas *fail-stop* e os manipuladores de erros propostos na norma MPI são os meios para informar a aplicação sobre as falhas de processos. A Figura 2 apresenta um exemplo com três processos (A, B e C) que realizam uma comunicação ponto-a-ponto.

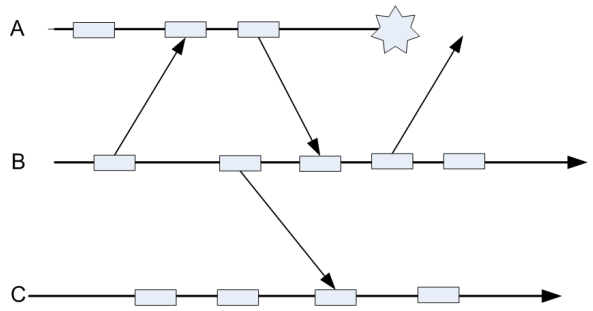


Figura 2. Detecção de falhas ULFM - Processo B detecta a falha em A.

Conforme apresenta a Figura 2, o processo B detecta a falha do processo A após enviar uma mensagem para A. No entanto, o processo C não identifica a falha em A. Essencialmente, a ocorrência de uma falha faz com que a comunicação não seja executada com sucesso. Por razões de desempenho, não há a notificação automática sobre a ocorrência de falhas. Dessa forma, é possível que somente alguns processos a identifiquem. Ao todo, a ULFM disponibiliza ao usuário cinco funções para lidar com as situações de falhas. Entre essas, algumas permitem estabelecer uma visão consistente entre os processos. As primitivas da ULFM são descritas a seguir.

A operação de revogação, `MPI_Comm_revoke`, notifica todos os processos que o comunicador MPI a que pertencem está inválido. Dessa forma, evita a inconsistência entre os processos associados a um comunicador. O comunicador torna-se inválido e as comunicações futuras, ou as comunicações pendentes, são interrompidas e marcadas com um código de erro.

A primitiva `MPI_Comm_agree` é empregada para determinar uma visão consistente entre os processos. Essa função executa uma operação coletiva e faz com que os

[5]http://www.open-mpi.org/faq/?category=ft e http://fault-tolerance.org/

[6]Um rascunho da nova versão, incluindo um capítulo para tolerância a falhas está disponível em https://svn.mpi-forum.org/trac/mpi-forum-web/attachment/ticket/323/mpi31-ticket323-r252-20140518.pdf

processos concordem com um valor lógico. Para fazer uso dessa primitiva o processo que identifica a falha deve antes revogar o comunicador. O construtor `MPI_Comm_shrink` permite à aplicação criar um novo comunicador, eliminando todos os processos falhos de um comunicador inválido. Essa operação é coletiva e executa um algoritmo de consenso para assegurar uma visão consistente no novo comunicador [Herault et al. 2015]. As primitivas `MPI_Comm_failure_ack` e `MPI_Comm_failure_get_acked` são usadas para informar quais processos dentro do comunicador se encontram falhos.

Por exemplo, na Figura 2, o processo B identifica a falha do processo A e então executa a função de revogação para que todos os processos corretos, no caso o processo C, marquem o seu comunicador inválido. Após isso, todos os processos executam a operação de acordo (`MPI_Comm_agree`) para garantir uma visão consistente sobre o estado do comunicador. Então, um novo comunicador válido, somente com processos corretos, pode ser criado por meio da função `MPI_Comm_shrink`. Se houver a necessidade de identificar qual processo falhou (no caso o A), as primitivas `MPI_Comm_failure_ack` e `MPI_Comm_failure_get_acked` podem ser usadas.

As falhas temporárias, tanto de rede quanto de processo, não fazem parte do escopo da ULFM, mas podem ser tratadas em nível de implementação. Uma falha temporária seria promovida a uma falha permanente (conforme o modelo *fail-stop*). Ou seja, se um processo sem-falha detecta que um processo deixa de responder, mesmo que temporariamente, o processo correto classifica esse processo como falho e continuamente ignora e descarta qualquer comunicação com o processo falho. Nesse caso, como dito anteriormente, para evitar que os processos tenham uma visão diferente sobre o estado de algum processo, as rotinas da ULFM (`MPI_Comm_revoke`, `MPI_Comm_agree` e `MPI_Comm_shrink`) podem ser usadas.

4. Algoritmo *Hyperquicksort*

Os algoritmos paralelos de ordenação são utilizados, por exemplo, em processamento de imagens, geometria computacional e teoria dos grafos [Quinn 2003]. Proposto por [Wagar 1987] o algoritmo *Hyperquicksort* combina a topologia e as características do hipercubo com a estratégia de ordenação empregada no *Quicksort* [Hoare 1962]. O algoritmo *Quicksort* é um dos algoritmos de ordenação sequenciais mais rápidos. É intuitivamente recursivo e baseia-se na comparação de chaves. Seu tempo de execução está estimado em $O(n \log n)$ no melhor e no médio caso e n^2 para o pior caso [Cormer et al. 2009].

O problema da ordenação no hipercubo consiste em empregar um conjunto P de processos, $P = \{p_0, p_1, ..., p_{2^{dim}-1}\}$, para ordenar uma lista K de números, $K = \{a_0, a_1, ..., a_{k-1}\}$. Os processos são organizados na forma de um hipercubo virtual de dimensão dim. Inicialmente os $|K|$ números são divididos igualmente entre os $|P|$ processos. Cada processo é responsável por ordenar uma lista de $\frac{|K|}{|P|}$ números. A ordenação é executada em rodadas, na quais os processos p_i e p_j trocam entre si parte da sua lista, determinada com base em um *número pivô* distribuído por um dos processos. Ao final de dim, rodadas de ordenação as listas estão ordenadas de forma que em cada processo p_i o maior número é menor ou igual ao menor número no processo $p_i + 1$, onde $0 \leq i \leq |P| - 2$. O algoritmo 1 a seguir apresenta o pseudocódigo do *Hyperquicksort*.

Inicialmente, cada processo ordena localmente a sua lista (linha 8). Os processos são organizados em *clusters* virtuais de tamanho regressivo (potência de 2) a cada rodada

Algorithm 1 Pseudocódigo do algoritmo *Hyperquicksort*

1: ***Hyperquicksort*** (para cada processo p executando em paralelo)
2: Initialization
3: $dim \leftarrow \log_2 |P|$ {*Dimensão do hipercubo*}
4: $rank \leftarrow process\_id$ {*Cada processo tem um valor único entre* $0..2^{dim} - 1$}
5: $lista \leftarrow K$ {*lista de números inicial em cada processo*}
6: $n \leftarrow |K|$ {*tamanho da lista de números em cada processo*}
7: Begin
8: quicksort($lista, n$)
9: **while** $dim > 0$ **do**
10: $cluster_i \leftarrow$ processes($rank, dim$)
11: $processo\_raiz \leftarrow$ root($rank, dim$)
12: **if** $rank == processo\_raiz$ **then**
13: $pivo \leftarrow$ medium($lista$)
14: broadcast($processo\_raiz, pivo, cluster_i$)
15: create_lists($higher\_list, lower\_list, list, pivo$)
16: $partner \leftarrow rank \oplus 2^{(dim-1)}$ {*ou exclusivo*}
17: **if** $rank > partner$ **then**
18: send($lower\_list, partner$)
19: receive($new\_higher_list, partner$
20: $list \leftarrow$ merge($higher\_list, new\_higher\_list$)
21: **else if** $rank < parter$ **then**
22: send($higher\_list, partner$)
23: receive($new\_lower\_list, partner$
24: $list \leftarrow$ merge($lower\_list, new\_lower\_list$)
25: $dim \leftarrow dim - 1$
26: quicksort($lista, n$)
End

de ordenação (linha 10). A Figura 3 representa o tamanho dos *clusters* para um hipercubo de 3 dimensões. Inicialmente, na primeira rodada de ordenação há oito processos agrupados em um único *clusters*. Para a segunda rodada, há dois *clusters* que agrupam quatro processos cada. Por último, há quatro *clusters* para cada dois processos. A partir de então, durante dim rodadas de ordenação o algoritmo executa os seguintes passos.

Os *clusters* são formados na respectiva rodada de ordenação (linha 10). O processo com menor *rank* em cada *cluster* é definido como o *processo raiz* (linha 11). Esse processo é o responsável por distribuir um número pivô aos demais processos do seu *cluster* (linhas 12-14). O número pivô é o número médio obtido a partir da sua lista de números (linha 13). Esse número pivô tem a mesma função da chave empregada no algoritmo *Quicksort*. Após receber o número pivô, cada processo divide a sua lista em duas outras listas: uma lista com números maiores que o número pivô e outra lista com os números menores que o número pivô (linha 15). Então, cada processo encontra um parceiro no seu *cluster* usando uma operação de ou exclusivo aplicada no seu próprio *rank* (linha 16). O processo de maior *rank* envia a sua lista de números menores que o número pivô ao seu parceiro (que possui *rank* maior) e recebe deste a lista com números maiores

que o número pivô (linhas 17-24). Após a troca, cada processo une a lista recebida com a lista que não foi trocada (linha 20 e 24) e realiza a ordenação nessa nova lista (linha 26).

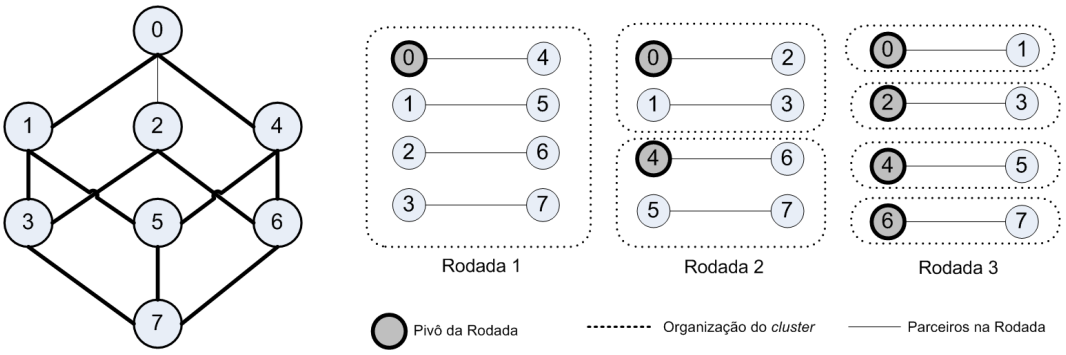


Figura 3. Algoritmo paralelo *Hyperquicksort* com 8 processos

A Figura 3 apresenta um exemplo de execução do algoritmo *Hyperquicksort* para 8 processos. A ordenação é realizada em 3 rodadas. Na primeira rodada, há um *cluster* com 8 processos e o processo raiz é o 0. O processo 0 distribui o seu número pivô aos demais processos do seu *cluster*. Os seguintes pares de processos são estabelecidos e trocam as suas listas de acordo com o número pivô recebido do processo 0: $(0, 4)$, $(1, 5)$, $(2, 6)$ e $(3, 7)$. Todos os processos reorganizam as suas listas e as ordenam localmente. Na segunda rodada de ordenação há dois *clusters* cada um com 4 processos. Os processos 0 e 4 são os processos raízes de seus respectivos *clusters*. Cada processo raiz envia o seu número pivô aos outros processos do seu *cluster*. Então as listas são trocadas entre os processos pares na rodada 2. Finalmente, na terceira rodada, todo o processo é repetido considerado os *clusters* e os processos em cada *cluster* de acordo com a terceira rodada de ordenação. A seguir, descreve-se a implementação tolerante a falhas do algoritmo *Hyperquicksort*.

4.1. *Hyperquicksort* Tolerante a Falhas

No início de cada rodada de ordenação, cada processo possui uma lista com os processos falhos. A partir de então, uma função de mapeamento é invocada em cada processo. Tal função auxilia a formar os pares de processos em uma rodada de ordenação e a definir qual processo sem-falha assume as tarefas do processo falho. Um processo p_i pode assumir até $n - 1$ processo falhos (no caso de somente um processo permanecer sem-falha). O processo p_i, além de executar normalmente as suas funções no algoritmo, deve também executar as funções que seriam executadas pelo processo falho. Cada processo p_j também precisa conhecer o processo p_i que assume as funções do processo falho. Para essas duas importantes tarefas da versão tolerante a falhas, a função $c_{i,s}$, definida no algoritmo *Hi-ADSD* [Duarte, Jr. and Nanya 1998], é utilizada para auxiliar no mapeamento. A função $c_{i,s}$ é descrita a seguir:

$$c_{i,s} = (i \oplus 2^{s-1}, c_{i\oplus 2^{s-1},1}, ..., c_{i\oplus 2^{s-1},s-1})$$

A função $c_{i,s}$ considera que os processos estão organizados logicamente em um hipercubo: i representa o processo p_i e s está relacionado a uma determinada rodada de ordenação. Inicialmente $s = dim$. O símbolo $\oplus$ representa a operação binária de OU exclusivo (XOR). A Tabela 1 apresenta um exemplo da função $c_{i,s}$ aplicada a um

Tabela 1. $c_{i,s}$ para um sistema com 8 nodos.

s	$c_{0,s}$	$c_{1,s}$	$c_{2,s}$	$c_{3,s}$	$c_{4,s}$	$c_{5,s}$	$c_{6,s}$	$c_{7,s}$
1	1	0	3	2	5	4	7	6
2	2 3	3 2	0 1	1 0	6 7	7 6	4 5	5 4
3	4 5 6 7	5 4 7 6	6 7 4 5	7 6 5 4	0 1 2 3	1 0 3 2	2 3 0 1	3 2 1 0

hipercubo de dimensão 3, ou seja, com 8 processos. O processo p_0 quando $s = 3$ tem o seguinte resultado: 4, 5, 6 e 7.

Os pares de processos são formados de acordo com o Algoritmo 2. Conforme apresenta o algoritmo, o parceiro de um processo p_i na rodada de ordenação s é o primeiro processo sem-falha na $c_{i,s}$. Portanto, para a primeira rodada de ordenação o processo p_0 deve trocar a sua lista com o primeiro processo sem-falha resultante de $c_{0,3}$. Se p_4 está sem-falha então p_0 e p_4 formam um par e trocam suas lista de acordo com as linhas 17-24 do Algoritmo 1 (considerando que p_0 também está sem-falha). Se p_4 se encontra falho então p_0 deve trocar a sua lista com p_5 e assim por diante. Se tanto p_5 e p_6 estiverem falhos então p_0 não troca a sua lista com nenhum processo nessa rodada de ordenação.

Algorithm 2 Função para encontrar um parceiro no cluster dim

```
1: function parceiro(rank, rodada)
2: Begin
3:     nodes ← c_{rank,rodada}
4:     j ← 0
5:     while j ≤ size(nodes) do
6:         if nodes[j] ∉ faults then
7:             return nodes[j]
8:         j ← j + 1
9:     return ⊥
   End
```

O processo que substitui um processo falho p_i em uma rodada de ordenação é escolhido de acordo com o Algoritmo 3. O substituto de um processo será o primeiro processo sem-falha de $c_{i,s}$, onde inicialmente $s = 1$ e i é o identificador do processo falho. Se não há processo sem-falha naquele *cluster*, s é incrementado até que um processo sem-falha seja encontrado. Considerando a primeira rodada de ordenação e os processos 0 e 4, se o processo p_4 está falho então p_5 assume as funções de p_4 na respectiva rodada de ordenação ($c_{4,1} = 5$ ver Tabela 1). Se p_5 também está falho então p_6 substitui p_4 pois é o primeiro processo sem-falha em $c_{4,2}$.

Um outro exemplo é fornecido a seguir. Suponha que o processo 2 está falho (Figura 3) na primeira rodada de ordenação. Essa rodada corresponde ao maior *cluster* ($s = 3$). Nessa rodada, os processos 2 e 6 forma um par quando ambos estão sem-falha. Entretanto, o primeiro processo sem-falha em $c_{6,3}$ é o processo 3, isto é, $c_{6,3} = 3$. O processo 3 é responsável pelo processo 2 pois é o primeiro nodo sem-falha de $c_{2,1}$. O processo 3 então se torna responsável pelas tarefas do processo 2. O processo 3 então realiza a leitura da lista de números do processo 2 e interage com o processo 6 substituindo o processo 2. Vale lembrar que o processo 3 também realiza normalmente a sua tarefa

Algorithm 3 Função para encontrar um parceiro no cluster dim

```
 1: function substituiFalho(rankFalho, dim)
 2: Begin
 3:     s ← 1
 4:     while s ≤ dim do
 5:         j ← 0
 6:         nodes ← c_{rankFalho,s}
 7:         while j ≤ size(nodes) do
 8:             if nodes[j] ∉ faults then
 9:                 return nodes[j]
10:             j ← j + 1
11:         s ← s + 1
12:     return ⊥
    End
```

com o processo 7, pois ambos estão sem-falha na rodada. Ao fim de cada rodada de ordenação cada processo salva a sua lista ordenada (linha 26 do Algoritmo 1) em disco compartilhado.

5. Implementação

A implementação foi realizada através das primitivas fornecidas na especificação ULFM. Vale lembrar que, por padrão, a detecção de falhas na ULFM é local, isto é, somente os processos que efetivamente se comunicam com o processo que falhou detectam a falha. No entanto, a ULFM permite, através das suas primitivas, implementar uma abordagem global de detecção de falhas, descrita a seguir.

Uma função chamada `detectaFalhos()` para detectar todos os processos que falharam foi implementada e inserida no começo de cada rodada de ordenação. Essa função inicia invocando a primitiva `MPI_Barrier` a fim de criar um ponto de sincronização entre os processos e verificar se houve falhas. Se ao menos um processo estiver falho, a função `MPI_Barrier` retorna um código de erro: `MPI_ERR_PROC_FAILED` ou `MPI_ERR_REVOKED`. A partir de então, todos os processos chamam uma função de acordo (`MPI_Comm_agree()`). A função `MPI_Comm_agree` executa uma operação coletiva entre os processos corretos no comunicador. No caso de falha, essa função notifica os processos que o comunicador está inválido. Na sequência o comunicador MPI é revogado usando a primitiva `MPI_Comm_revoke()`.

A partir de então são empregadas as rotinas `MPI_Comm_failure_ack()` e `MPI_Comm_failure_get_acked()` para identificar quais processos dentro do comunicador estão falhos. Após isso, a rotina `MPI_Comm_shrink()` cria um novo comunicador, eliminando todos os processos que falharam. Operações de grupo de processos em MPI são ainda realizadas para manter os processos no novo comunicador com o mesmo *rank* que tinham antes da falha.

Um vetor com o estado (falho ou sem-falha) dos processos é mantido em cada processo usando a função `detectaFalhos()`. Uma vez que cada processo possui a lista de processos falhos, as funções apresentadas nos Algoritmos 2 e 3 são utilizadas para permitir que os processos sem-falha continuem a execução.

Em relação as listas de números de posse dos processos falhos duas abordagens foram implementadas: 1) os processos sem-falha mantém a lista em nome de cada processo falho e; 2) os processos sem-falha incorporam a lista do processo falho à sua lista. Na primeira possibilidade se há 8 processos, então ao final haverá 8 listas de números, mesmo se $n-1$ processos falharem. Essa versão permite - caso um processo pudesse retornar após uma falha -, que um processo que deixou de participar de uma rodada de ordenação específica retorne em uma rodada posterior. Por exemplo, o processo 0 participa somente da primeira e da última rodada de ordenação, supondo 3 rodadas de ordenação. Na segunda possibilidade, o número de listas será igual ao número de processos que não falharam. Os resultados a seguir apresentam o desempenho considerando a primeira abordagem.

Uma função `injetaFalhas()` foi codificada para injetar falhas durante a execução. Cada processo recebe o número total de processos que devem falhar e o número total de rodadas de ordenação. A partir de então, a função aleatoriamente define em qual rodada um determinado processo deve falhar. Um processo pode vir a se tornar falho em qualquer rodada de ordenação. Um processo finaliza a si mesmo através do sinal `SIGKILL` se seu identificador havia sido sorteado para falhar naquela rodada. Uma vez que os processos executam a função `injetaFalhas()` em paralelo a mesma semente é utilizada na função `srand()` em cada processo.

6. Resultados

Os experimentos foram executados no sistema operacional *Linux Kernel* 3.2.0 em 16 processadores AMD *Opteron* com 2.400 MHz. A rede do laboratório é *Ethernet* de 100 Mbps. Para todos os resultados apresentados cada experimento foi repetido 30 vezes; são apresentadas a média e intervalo de confiança de 95%. O código MPI foi escrito em liguagem C. A biblioteca MPI utilizada foi a Open MPI versão 1.7 estendida com a ULFM (1.7ft_b4)[7].

São apresentados resultados de desempenho para quatro cenários: 1) sem-falhas; 2) somente um processo falho; 3) metade dos processos falham; 4) $n-1$ processos falham. O objetivo não é apresentar o *speedup*, mas a capacidade do algoritmo de se manter em execução mesmo perante falhas. As falhas são inseridas no início de cada rodada de ordenação.

A Figura 4 apresenta o desempenho do *Hyperquicksort* tolerante a falhas para ordenar 1 bilhão de números inteiros usando 16 processos MPI e aplicando os quatro cenários de falhas. O tempo de execução do algoritmo sem falhas é de aproximadamente 420 segundos e a variação no desempenho é pequena.

No cenário com uma falha, o tempo de execução do algoritmo é ligeiramente menor. Isso se deve ao momento em que a falha ocorre. Uma falha que acontece logo na primeira rodada de ordenação prejudica mais o desempenho do algoritmo do que uma falha que ocorre na última rodada de ordenação. Uma falha na primeira rodada obriga o algoritmo a se reconfigurar logo no começo, fazendo um processo acumular tarefas desde o início. Por outro lado, uma falha na última rodada de ordenação faz com o processo que substitui o processo falho economize uma troca de listas: o processo substituto é o processo parceiro do processo falho. Por exemplo, supondo que na última rodada de

[7]http://fault-tolerance.org/ulfm/downloads/

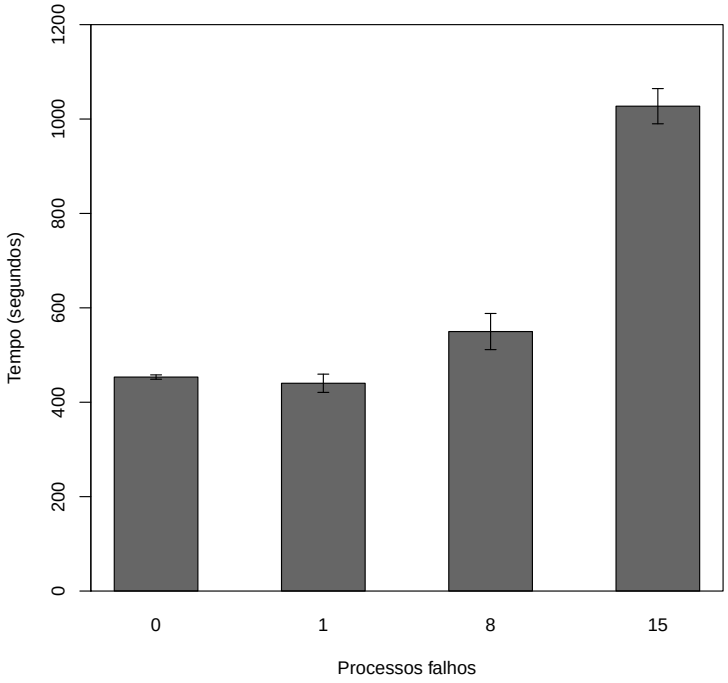


Figura 4. Desempenho de 16 processos perante falhas ordenando 1.024 x 10^6 números inteiros.

ordenação o processo 1 falhe (Figura 3, rodada 3). O processo 0 é o processo que substitui o processo falho, mas o processo 0 também é o parceiro do processo falho. Dessa forma, o processo substituto lê a lista de números do processo falho e executa as operações previstas sem usar as primitivas `MPI_Send()` e `MPI_Recv`. Outra situação que pode aumentar o desempenho é a falha do processo raiz. Lembrando que o processo raiz é o responsável por distribuir o seu número pivô os demais processos do seu *cluster*.

Para o cenário com metade das falhas, o desempenho é prejudicado em cerca de 30%. Para esse cenário, houve casos em que apenas um processo falhou na primeira rodada de ordenação, dois processos falharam na segunda rodada de ordenação, três processos falharam na terceira rodada de ordenação e dois processos falharam na última rodada de ordenação. O cenário com $n - 1$ falhas apresenta perto de 150% de sobrecarga. No entanto, para todos os casos, a ordenação foi realizada com sucesso, apesar das falhas.

7. Conclusão

Este trabalho apresentou uma implementação MPI do algoritmo de ordenação paralela *Hyperquicksort* que tolera até $n - 1$ falhas. A implementação empregou a especificação ULFM para detectar uma falha local. A partir de então uma função foi desenvolvida para permitir que todos os processos que participam da computação conheçam a falha. Uma função de mapeamento dos processos foi implementada para permitir que os processos sem-falha substituam os processos falhos. O algoritmo tolerante a falhas foi executado para ordenar 1 bilhão de números inteiros. Resultados apresentam que o algoritmo foi capaz de continuar a sua execução apesar das falhas de processos.

Na avaliação realizada, as falhas foram inseridas no início de uma rodada de ordenação. Um simples extensão no algoritmo o permitiria lidar com uma falha em qualquer momento da ordenação: nesse caso os processos que não falharam reiniciariam a ordenação a partir da rodada anterior.

Tradicionalmente, as implementações MPI abortam toda a sua execução mesmo se um único processo falhar. A ULFM delega ao programador da aplicação a tarefa de lidar com as falhas. Entre essas tarefas está escolher a estratégia de recuperação que melhor se adapta a sua aplicação. Um trabalho futuro é investigar se a mesma estratégia de recuperação empregada no *Hyperquicksort* pode ser aplicada em outros algoritmos de ordenação baseados no hipercubo. Além disso, é possível investigar se outras classes de aplicações MPI, para além dos algoritmos de ordenação, podem adotar a mesma abordagem de recuperação aplicada ao *Hyperquicksort*. Uma API de programação pode ser projetada para permitir que o programador insira de forma transparente a estratégia de recuperação nas aplicações MPI.

Referências

Batchu, R., Dandass, Y. S., Skjellum, A., and Beddhu, M. (2004). MPI/FT: A model-based approach to low-overhead fault tolerant message-passing middleware. *Cluster Computing*, 7(4):303–315.

Bland, W., Bosilca, G., Bouteiller, A., Herault, T., and Dongarra, J. (2012a). A proposal for user-level failure mitigation in the mpi-3 standard. Technical report, Department of Electrical Engineering and Computer Science, University of Tennessee.

Bland, W., Bouteiller, A., Hérault, T., Bosilca, G., and Dongarra, J. (2013). Post-failure recovery of MPI communication capability: Design and rationale. *International Journal of HPC Applications*, 27(3):244–254.

Bland, W., Bouteiller, A., Hérault, T., Hursey, J., Bosilca, G., and Dongarra, J. J. (2012b). An evaluation of user-level failure mitigation support in MPI. In *EuroMPI*, volume 7490 of *LNCC*, pages 193–203. Springer.

Bland, W., Du, P., Bouteiller, A., Hérault, T., Bosilca, G., and Dongarra, J. (2012c). A checkpoint-on-failure protocol for algorithm-based recovery in standard MPI. In *Euro-Par*.

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

Chen and Chung (2001). Improved fault-tolerant sorting algorithm in hypercubes. *TCS: Theoretical Computer Science*, 255.

Cormer, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introductions to Algorithms'*. The MIT Press.

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

Fagg, G. E. and Dongarra, J. (2000). FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In *Recent advances in PVM-MPI*, LNCS. Springer.

Freiling, F. C., Guerraoui, R., and Kuznetsov, P. (2011). The failure detector abstraction. *ACM Comput. Surv.*, 43(2):9:1–9:40.

Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S. (2004). Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary.

Gropp, W., Lusk, E., Doss, N., and Skjellum, A. (1996). A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828.

Gropp, W. and Lusk, E. L. (2004). Fault tolerance in message passing interface programs. *International Journal of HPC Applications*, 18(3):363–372.

Herault, T., Bouteiller, A., Bosilca, G., Gamell, M., Teranishi, K., Parashar, M., and Dongarra, J. (2015). Practical scalable consensus for pseudo-synchronous distributed systems. In *SuperComputing conference*.

Hoare, C. A. R. (1962). Quicksort. *The Computer Journal*, 5(4):10–15.

Hursey, J., Graham, R. L., Bronevetsky, G., Buntinas, D., Pritchard, H., and Solt, D. G. (2011). Run-through stabilization: An MPI proposal for process fault tolerance. In *EuroMPI*.

Johnsson (1984). Combining parallel and sequential sorting on a boolean n-cube. In *ICPP: 13th International Conference on Parallel Processing*.

Leighton, F. T. (1991). *Introduction to Parallel Algorithms and Architectures: Array, Trees, Hypercubes*. Morgan Kaufmann Publishers.

MPI Forum (2015). Document for a standard message-passing interface 3.1. Technical report, University of Tennessee, http://www.mpi-forum.org/docs/mpi-3.1.

Parhami, B. (1999). *Introduction to Parallel Processing: Algorithms and Architectures*. Kluwer Academic Publishers, Norwell, MA, USA.

Plaxton, C. G. (1989). Load balancing, selection sorting on the hypercube. In *SPAA*, pages 64–73.

Quinn, M. J. M. J. (2003). *Parallel programming in C with MPI and OpenMP*. McGraw-Hill, pub-MCGRAW-HILL:adr.

Seidel, S. R. and George, W. L. (1988). Binsorting on hypercubes with d-port communication. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications - Volume 2*, C3P, pages 1455–1461, New York, NY, USA. ACM.

Sheu, Chen, and Chang (1992). Fault-tolerant sorting algorithm on hypercube multicomputers. In *ICPP: 21th International Conference on Parallel Processing*.

Suo, G., Lu, Y., Liao, X., Xie, M., and Cao, H. (2013). Nr-mpi: A non-stop and fault resilient mpi. In *ICPADS*.

Wagar, B. (1987). Hyperquicksort: A fast sorting algorithm for hypercubes. *Hypercube Multiprocessors*, 1987:292–299.

Won, Y. and Sahni, S. (1988). A balanced bin sort for hypercube multicomputers. *The Journal of Supercomputing*, 2(4).

Uma Proposta de Difusão Confiável Hierárquica em Sistemas Distribuídos Assíncronos

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

[1] Sorbonne Universités, UPMC Université. Paris 06
CNRS, Inria, LIP6 – Place Jussieu, 4 – 75005 – Paris, France

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

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

denis.jeanneau@lip6.fr, luiz.rodrigues@unioeste.br

Abstract. *This paper presents the work in progress on a hierarchical reliable broadcast solution based on the VCube virtual topology that assumes an asynchronous system. This topology is built and dynamically adapts itself with information obtained from an underlying monitoring system. Broadcast messages are disseminated through a spanning tree that is created and dynamically maintained embedded on a VCube. Processes fail by crashing and a fault is assumed to be eventually detected by all correct processes. In particular we discuss how to deal with false suspicions that arise in the asynchronous environment.*

Resumo. *Este trabalho apresenta a versão preliminar de uma solução hierárquica para a difusão confiável de mensagens com base na topologia virtual mantida pelo VCube. A topologia é construída e adaptada dinamicamente com base nas informações de falhas obtidas de um sistema não confiável de monitoramento. As mensagens são propagadas por uma árvore geradora criada dinamicamente sobre os enlaces mantidos pelo VCube. Os processos podem falhar por crash sem recuperação e uma falha é detectada por todos os processos corretos em um tempo finito. Mensagens diferenciadas são utilizadas para tratar falsas suspeitas geradas pela execução em ambiente assíncrono.*

1. Introdução

Um processo em um sistema distribuído utiliza difusão para enviar uma mensagem a todos os outros processos do sistema [Bonomi et al. 2013]. No entanto, se este processo falha durante o procedimento de difusão, alguns processos podem receber a mensagem enquanto outros não. A difusão confiável garante que, mesmo após a falha do emissor, todos os processos corretos recebem a mensagem difundida por ele [Hadzilacos e Toueg 1993].

Algoritmos de difusão tolerante a falhas são normalmente implementados utilizando enlaces ponto-a-ponto confiáveis e primitivas SEND e RECEIVE. Os processos invocam BROADCAST(m) e DELIVER(m) para difundir e receber uma mensagem m para/de outros processos da aplicação, respectivamente. Para incluir tolerância a falhas, um detector de falhas [Chandra et al. 1996] pode ser utilizado para notificar o algoritmo de *broadcast*, que deve reagir apropriadamente quando uma falha é detectada.

Este trabalho apresenta uma proposta de algoritmo de difusão confiável no qual cada processo é alcançado por meio de uma árvore dinâmica construída sobre uma topologia de hipercubo virtual chamada VCube [Duarte et al. 2014]. O sistema é representado por um grafo completo com enlaces confiáveis. Os processos do sistema são organizados em *clusters* progressivamente maiores formando um hipercubo completo quando não há processos falhos. Em caso de suspeitas, os processos considerados corretos são reconectados entre si para manter as propriedades logarítmicas do hipercubo.

O restante do texto está organizado nas seguintes seções. A Seção 2 discute os trabalhos relacionados. A Seção 3 apresenta as definições básicas, o modelo do sistema e o VCube. O algoritmo de *broadcast* confiável proposto é apresentado na Seção 4. A Seção 5 apresenta a conclusão e os trabalhos futuros.

2. Trabalhos Relacionados

Grande parte dos algoritmos de difusão são baseados em árvores geradoras. Schneider et al. (1984) introduziram um algoritmo de difusão tolerante a falhas baseado em árvore no qual a raiz é o processo que inicia a transmissão, ou seja, o remetente. Cada nodo, incluindo o remetente, envia a mensagem para todos os seus sucessores na árvore. Se um processo p que pertence à árvore falhar, outro processo assume a responsabilidade de retransmitir as mensagens que p deveria ter transmitido se estivesse correto. Os processos podem falhar por *crash* e a falha de um processo é detectada por um módulo de detecção de falhas após um intervalo finito, mas não conhecido. Um processo pode enviar uma próxima mensagem somente após a difusão anterior ter sido concluída. No entanto, os autores não descrevem como a detecção de falhas é implementada, tão pouco fornecem um algoritmo para construir e reorganizar a árvore após a falha.

Em Wu (1996), os autores apresentam um algoritmo de difusão tolerante a falhas para hipercubos baseado em árvores binomiais. O algoritmo pode recursivamente regenerar uma subárvore falha, induzida por um nodo com defeito, através de uma das folhas da árvore. No entanto, ao contrário da abordagem proposta neste trabalho, a solução exige o bloqueio do sistema até que a árvore seja reconstruída.

Liebeherr e Beam (1999) apresentam um procolo, chamado HyperCast, que organiza os membros de um grupo *multicast* em um hipercubo lógico usando o código *Gray* para ordená-los. A árvore é sobreposta no hipercubo para evitar implosão de ACKs. O processo com o identificar mais alto é considerado a raiz da árvore. Entretanto, em função de falhas, múltiplos nodos podem considerar a si próprios como raiz e/ou diferentes nodos podem ter visões diferentes sobre a identidade da raiz.

Em Rodrigues et al. (2014) foi apresentada uma solução para *broadcast* confiável utilizando árvores dinâmicas no VCube. O algoritmo permite a propagação de mensagens utilizando múltiplas árvores construídas dinamicamente a partir de cada emissor e que incluem todos os nodos do sistema. Diferente da solução proposta neste trabalho, o modelo é síncrono e o detector de falhas é perfeito.

3. Definições e Modelo do Sistema

Considera-se um sistema distribuído como um conjunto finito P com $n > 1$ processos $\{p_0, .., p_{n-1}\}$ que se comunicam por troca de mensagens. A rede é representada por um grafo completo. No entanto, processos são organizados em uma topologia de hipercubo

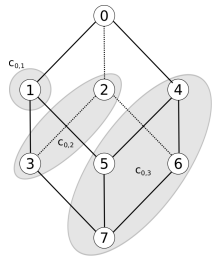
virtual, chamada VCube [Ruoso 2013]. As operações de envio e recebimento são atômicas e os enlaces são confiáveis. O sistema admite falhas de *crash* permanente. Um processo que nunca falha é considerado *correto* ou *sem-falha*. Caso contrário, ele é dito *falho* ou *suspeito*. Considera-se que o VCube implementa um detector de falhas $\diamond S$, isto é, processos falhos são permanentemente suspeitos, mas podem ocorrer falsas suspeitas.

3.1. O VCube

Cada processo que executa o VCube é capaz de testar outros processos no sistema para verificar se estão corretos ou falhos. Os processos são organizados em *clusters* progressivamente maiores. Cada *cluster* $s = 1, .., \log_2 n$ possui 2^s elementos, sendo n o total de processos no sistema. Para cada rodada um processo i testa o primeiro processo sem-falha j na lista de processos de cada *cluster* s e obtém dele as informações que ele possui sobre os demais processos do sistema.

Os membros de cada *cluster* s e a ordem na qual eles são testados por um processo i são obtidos da lista gerada pela função $c_{i,s} = (i \oplus 2^{s-1}, c_{i\oplus 2^{s-1},1}, ..., c_{i\oplus 2^{s-1},s-1})$ ($\oplus$ é a operação binária de OU exclusivo – XOR).

A Figura 1 exemplifica a organização hierárquica dos processos em um hipercubo de três dimensões com $n = 2^3$ elementos. A tabela da direita apresenta os elementos de cada *cluster* $c_{i,s}$. Como exemplo, na primeira rodada o processo p_0 testa o primeiro processo no *cluster* $c_{0,1} = (1)$ e obtém informações sobre o estado dos demais processos armazenada em p_1. Em seguida, p_0 testa o processo p_2, que é primeiro processo no *cluster* $c_{0,2} = (2, 3)$. Por fim, p_0 executa testes no processo p_4 do *cluster* $c_{0,3} = (4, 5, 6, 7)$. Como cada processo executa estes procedimentos de forma concorrente, ao final da última rodada todo processo será testado ao menos uma vez por um outro processo. Isto garante uma latência de diagnóstico máxima de $\log_2^2 n$ rodadas.



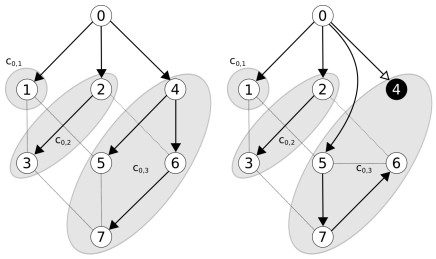
s	$c_{0,s}$	$c_{1,s}$	$c_{2,s}$	$c_{3,s}$	$c_{4,s}$	$c_{5,s}$	$c_{6,s}$	$c_{7,s}$
1	1	0	3	2	5	4	7	6
2	2 3	3 2	0 1	1 0	6 7	7 6	4 5	5 4
3	4 5 6 7	5 4 7 6	6 7 4 5	7 6 5 4	0 1 2 3	1 0 3 2	2 3 0 1	3 2 1 0

Figura 1. Organização Hierárquica do VCube de $d = 3$ dimensões.

4. O Algoritmo de Difusão Confiável Proposto

Seja i um processo que executa o algoritmo de difusão confiável e $d = \log_2 n$ a dimensão do d-VCube com 2^d processos. O Algoritmo 1 apresenta o pseudo-código da solução de difusão confiável proposta.

A função $cluster_i(j) = s$ calcula o identificador s do *cluster* do processo i que contém o processo j, $1 \leq s \leq d$. Por exemplo, considerando o 3-VCube da Figura 1, $cluster_0(1) = 1$, $cluster_0(2) = cluster_0(3) = 2$ e $cluster_0(4) = cluster_0(5) = cluster_0(6) = cluster_0(7) = 3$. Três tipos de mensagens são utilizados: $\langle TREE, m\rangle$ para identificar a mensagem de aplicação m que está sendo propagada na árvore; $\langle DELV, m\rangle$ para as mensagens enviadas aos processos considerados falhos,



(a) Sem falhas (b) p_4 é suspeito

Figura 2. Difusão confiável no processo 0 (p_0)

evitando que falsas suspeitas impliquem em não recebimento por processos corretos; e $\langle ACK, m\rangle$ para confirmar o recebimento de m pelo destinatário que recebe TREE. Cada mensagem m contém ainda dois parâmetros: (1) o identificador da origem, isto é, o processo que iniciou a difusão, obtido com a função $source(m)$; e (2) o *timestamp*, um contador sequencial local que identifica de forma única cada mensagem gerada em um processo, obtido pela função $ts(m)$. Um processo obtém as informações sobre o estado dos demais processos pelo algoritmo VCube.

As variáveis locais mantidas pelos processos são:

- $correct_i$: conjunto dos processos considerados corretos pelo processo i;
- $last_i[n]$: a última mensagem recebida de cada processo fonte;
- ack_set_i: o conjunto de ACKs pendentes no processo i. Para cada mensagem $\langle TREE, m\rangle$ (re)-transmitida por i de um processo j para um processo k, um elemento $\langle j, k, m\rangle$ é adicionado a este conjunto. O símbolo $\perp$ representa um elemento nulo. O asterisco é usado como curinga. Por exemplo, um elemento $\langle j, *, m\rangle$, por exemplo, representa todos os ACKs pendentes para uma mensagem m recebida pelo processo j e retransmitida para qualquer outro processo.
- $pending_i$: lista de mensagens m recebidas pelo processo i de um processo fonte $source(m)$ que ainda não podem ser entregues à aplicação por estarem fora de ordem, isto é, $ts(m) > ts(last_i(source(m)) + 1$.

Um processo i inicia a difusão invocando o método BROADCAST(m). A linha 7 garante que um novo *broadcast* só é iniciado após o término do anterior, isto é, quando não há *acks* pendentes para a mensagem $last_i[i]$. Nas linhas 9 - 10 a nova mensagem m é entregue localmente (propriedade de entrega confiável) e em seguida enviada a todos os vizinhos no VCube através da função BROADCAST_TREE. Esta função utiliza BROADCAST_CLUSTER para enviar mensagens TREE para cada primeiro processo sem-falha em cada cluster $s = 1..\log_2 n$, e mensagens DELV para os processos considerados falhos (suspeitos). Para cada mensagem TREE enviada, um *ack* é incluído na lista de *acks* pendentes. A Figura 2 ilustra exemplos para um VCube de 3 dimensões. A Figura 2(a) mostra uma execução sem falhas considerando o processo 0 (p_0) como fonte. Após fazer a entrega local, p_0 envia uma cópia da mensagem para p_1, p_2 e p_4, que são os vizinhos dele em cada *cluster*. Embora p_5 também pertença ao mesmo *cluster* de p_4, p_0 não envia uma cópia da mensagem para p_5, pois não há aresta no VCube conectando os dois processos.

Quando um processo i recebe uma mensagem $\langle TREE, m\rangle$ de um processo j (linha 39) ele invoca HANDLE_MESSAGE. Nela é verificado se a mensagem é nova comparando os *timestamps* da última mensagem armazenada em $last_i[j]$ e da mensagem recebida m (linha 33), garantindo a propriedade de integridade. Se m é um nova mensagem,

$last_i[j]$ é atualizado e a mensagem é entregue à aplicação. Em seguida, o processo i verifica se o processo origem da mensagem está falho. Se a origem ainda é considerada correta, m é retransmitida para os vizinhos em cada *cluster* interno ao *cluster* de i que também fazem parte do grupo com os destinatários da mensagem. Isso é feito durante a execução da função BROADCAST_CLUSTER com parâmetro $h = cluster_i(j) - 1$. A Figura 2(a) ilustra a propagação feita por p_4 para p_5. Se i é uma folha da árvore (*clusters* $s = 1$) ou se não existe vizinho correto pertencente aos destinatários, nenhum *ack* pendente é adicionado ao conjunto ack_set_i e CHECKACKS envia uma mensagem `ACK` para j. Por outro lado, se um processo i recebe uma mensagem nova $\langle TREE, m\rangle$, mas verifica que $source(m)$ foi detectado como falho, o processo de *broadcast* é reiniciado considerando a árvore com raiz em i.

Quando uma mensagem $\langle ACK, m\rangle$ é recebida, o conjunto ack_set_i é atualizado e, se não existem mais *acks* pendentes para a mensagem m, CHECKACKS envia um `ACK` para o processo k do qual i recebeu a mensagem `TREE` anteriormente. No entanto, se $k = i$, a mensagem `ACK` alcançou o processo fonte ou o processo que retransmitiu a mensagem após a falha do processo fonte. Nesse caso, a mensagem de `ACK` não precisa mais ser propagada.

A detecção de um processo falho j é tratada após a notificação do evento

Algoritmo 1 Difusão confiável no processo i

```
1: last_i[n] ← {⊥, ..., ⊥}
2: ack_set_i ← ∅
3: correct_i ← {0, ..., n − 1}
4: pending_i ← ∅

5: procedure BROADCAST(message m)
6:     if source(m) = i then
7:         wait until ack_set_i ∩ {⟨⊥, ∗, last_i[i]⟩} = ∅
8:         last_i[i] ← m
9:         DELIVER(m)
10:    BROADCAST_TREE(⊥, m, log_2 n)

11: procedure BROADCAST_TREE(process j, message m, integer h)
12:    for all s ∈ [1, h] do
13:        BROADCAST_CLUSTER(j, m, s)

14: procedure BROADCAST_CLUSTER(process j, message m, integer s)
15:    sent ← false
16:    for all k ∈ c_{i,s} do
17:        if sent = false then
18:            if ⟨j, k, m⟩ ∈ ack_set_i and k ∈ correct_i then
19:                sent ← true
20:            else if k ∈ correct_i then
21:                SEND(⟨TREE, m⟩) to p_k
22:                ack_set_i ← ack_set_i ∪ {⟨j, k, m⟩}
23:                sent ← true
24:            else if ⟨j, k, m⟩ ∉ ack_set_i then
25:                SEND(⟨DELV, m⟩) to p_k

26: procedure CHECK_ACKS(process j, message m)
27:    if j ≠ ⊥ and ack_set_i ∩ {⟨j, ∗, m⟩} = ∅ then
28:        if j ∈ correct_i then
29:            SEND(⟨ACK, m⟩) to p_j

30: procedure HANDLE_MESSAGE(message m)
31:    pending_i ← pending_i ∪ {m}
32:    while ∃l ∈ pending_i : (source(l) = source(m) ∧
       ts(l) = ts(last_i[source(m)]) + 1)
33:        or (last_i[source(m)] = ⊥ ∧ ts(l) = 0) do
34:        last_i[source(m)] ← l
35:        pending_i ← pending_i \ {l}
36:        DELIVER(l)
37:    if source(m) ∉ correct_i then
38:        BROADCAST(m)

39: upon receive ⟨TREE, m⟩ from p_j
40:    HANDLE_MESSAGE(m)
41:    BROADCAST_TREE(m, cluster_i(j) − 1)
42:    CHECK_ACKS(j, m)

43: upon receive ⟨DELV, m⟩ from p_j
44:    HANDLE_MESSAGE(m)

45: upon receive ⟨ACK, m⟩ from p_j
46:    for all k = x : ⟨x, j, m⟩ ∈ ack_set_i do
47:        ack_set_i ← ack_set_i \ {⟨k, j, m⟩}
48:        CHECK_ACKS(k, m)

49: upon notifying crash(process j)
50:    correct_i ← correct_i ∖ {j}
51:    for all p = x, m = y : ⟨x, j, y⟩ ∈ ack_set_i ∩
       {⟨∗, j, ∗⟩} do
52:        BROADCAST_CLUSTER(p, m, cluster_i(j))
53:        ack_set_i ← ack_set_i \ {⟨p, j, m⟩}
54:        CHECK_ACKS(p, m)
55:    if last_i[j] ≠ ⊥ then
56:        BROADCAST(last_i[j])

57: upon notifying up(process j)
58:    correct_i ← correct_i ∪ {j}
```

CRASH(j). Três ações são realizadas: (1) atualização da lista de processos corretos; (2) remoção dos *acks* pendentes que contém o processo j como destino ou aqueles em que a mensagem m foi originada em j; (3) reenvio das mensagens anteriormente transmitidas ao processo j para o novo vizinho correto k no mesmo *cluster* de j, se existir um. Esta retransmissão desencadeia uma propagação na nova estrutura da árvore. No exemplo da Figura 2(b), após a notificação de falha de p_4, p_0 retransmite a mensagem para o processo p_5, visto que $c_{0,3} = (4, 5, 6, 7)$, p_5 é o próximo processo sem falha no *cluster* $s = 3$. A propagação continua pelos demais processos corretos do *cluster*, isto é, p_6 e p_7. Se p_4 é considerado falho antes do início da difusão, uma mensagem DELV é enviada a ele para garantir o recebimento por p_4 em caso de falsa suspeita.

5. Conclusão

Este trabalho apresentou uma proposta de solução distribuída para a difusão confiável (*broadcast*) em sistemas distribuídos sujeitos a falhas de *crash* em ambientes assíncronos. Árvores com raiz em cada processo são construídas e mantidas dinamicamente sobre uma topologia de hipercubo virtual denominada VCube. Em caso de falhas, os processos são reorganizados de forma a manter as propriedades logarítmicas do hipercubo. Falsas suspeitas são contornadas pelo envio de mensagens adicionais aos processos considerados falhos, porém mantendo-se as principais propriedades do hipercubo.

Como trabalhos futuros, o algoritmo será especificado formalmente, incluindo provas de correção, e testes de desempenho serão realizados por simulação comparando-o com outras soluções.

Referências

Bonomi, S., Del Pozzo, A. e Baldoni, R. (2013). Intrusion-tolerant reliable broadcast. Technical report, Sapienza Università di Roma,.

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

Duarte, Jr., E. P., Bona, L. C. E. e Ruoso, V. K. (2014). VCube: A provably scalable distributed diagnosis algorithm. In: *5th Work. on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA'14, pp. 17–22, Piscataway, USA. IEEE Press.

Hadzilacos, V. e Toueg, S. (1993). Fault-tolerant broadcasts and related problems. In: Distributed systems, pp. 97–145. ACM Press, New York, NY, USA, 2 ed.

Liebeherr, J. e Beam, T. (1999). HyperCast: A protocol for maintaining multicast group members in a logical hypercube topology. In: Rizzo, L. e Fdida, S., editores, *Networked Group Communication*, v. 1736 de *LNCS*, pp. 72–89. Springer Berlin Heidelberg.

Rodrigues, L. A., Duarte Jr., E. P. e Arantes, L. (2014). Árvores geradoras mínimas distribuídas e autonômicas. In: *XXXII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, SBRC'14.

Ruoso, V. K. (2013). Uma estratégia de testes logarítmica para o algoritmo Hi-ADSD. Dissertação de Mestrado, Universidade Federal do Paraná.

Schneider, F. B., Gries, D. e Schlichting, R. D. (1984). Fault-tolerant broadcasts. *Sci. Comput. Program.*, 4(1):1–15.

Wu, J. (1996). Optimal broadcasting in hypercubes with link faults using limited global information. *J. Syst. Archit.*, 42(5):367–380.

REALIZAÇÃO

computação UFBA

UFBA 70 ANOS

INSTITUTO FEDERAL BAHIA

PROMOÇÃO

LARC

SBC Sociedade Brasileira de Computação

FOMENTO

CNPq Conselho Nacional de Desenvolvimento Científico e Tecnológico

CAPES

APOIO

STI 40 Superintendência de Tecnologia da Informação | UFBA

POP-BA

PRONEX Filosofia e Ciências

FUNDAÇÃO Pierre Verger

RNP REDE NACIONAL DE ENSINO E PESQUISA

PATROCÍNIO DIAMANTE

nic.br Núcleo de Informação e Coordenação do Ponto BR

cgi.br Comitê Gestor da Internet no Brasil

PATROCÍNIO OURO

acm sigcomm

L3Networks

SOPHOS · arcserve

LogMeIn · KASPERSKY

GRUPO BINÁRIO Integração e Serviços

aruba a Hewlett Packard Enterprise company

PATROCÍNIO BRONZE

Google

IMTECH

Solutis tecnologias

PROTELE engenharia