SBRC 2014

Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos

Anais

XV Workshop de Testes e Tolerância a Falhas

WTF 2014

Florianópolis - SC
5 a 9 de Maio de 2014

XXXII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos

5 a 9 de Maio de 2014
Florianópolis - SC

Anais

XV Workshop de Testes e Tolerância a Falhas (WTF 2014)

Editora
Sociedade Brasileira de Computação (SBC)

Organizadores
Luciana Arantes (Université de Paris 6)
Carlos André Guimarães Ferraz (UFPE)
Joni da Silva Fraga (UFSC)
Frank Siqueira (UFSC)

Realização
Universidade Federal de Santa Catarina (UFSC)

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

Copyright ©2014 da Sociedade Brasileira de Computação
Todos os direitos reservados

Capa: Vanessa Umbelino (PostMix)
Produção Editorial: Roberto Willrich (UFSC)

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 (15: 2014: Florianópolis, SC)

Anais / Anais do XV Workshop de Testes e Tolerância a Falhas; organizado por Luciana Arantes... [et al.] - Porto Alegre: SBC, c2014

202 p.

WTF 2014
Realização: Universidade Federal de Santa Catarina
ISSN: 2177-496X

1. Redes de Computadores - Congressos. 2. Sistemas Distribuídos- Congressos. I. Arantes, Luciana. II. Sociedade Brasileira de Computação. III. Título.

Promoção

Sociedade Brasileira de Computação (SBC)

Diretoria

Presidente
Paulo Roberto Freire Cunha (UFPE)

Vice-Presidente
Lisandro Zambenedetti Granville (UFRGS)

Diretora Administrativa
Renata de Matos Galante (UFRGS)

Diretor de Finanças
Carlos André Guimarães Ferraz (UFPE)

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

Diretora de Educação
Mirella Moura Moro (UFMG)

Diretor de Publicações
José Viterbo Filho (UFF)

Diretora de Planejamento e Programas Especiais
Claudia Lage Rebello da Motta (UFRJ)

Diretor de Secretarias Regionais
Marcelo Duduchi Feitosa (CEETEPS)

Diretor de Divulgação e Marketing
Edson Norberto Caceres (UFMS)

Diretor de Relações Profissionais
Roberto da Silva Bigonha (UFMG)

Diretor de Competições Científicas
Ricardo de Oliveira Anido (UNICAMP)

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

Diretor de Articulação de Empresas
Avelino Francisco Zorzo (PUC-RS)

Promoção

Sociedade Brasileira de Computação (SBC)

Conselho

Mandato 2013-2017

Alfredo Goldman (IME/USP)
José Palazzo Moreira de Oliveira (UFRGS)
Maria Cristina Ferreira de Oliveira (ICMC/USP)
Thais Vasconcelos Batista (UFRN)
Wagner Meira Junior (UFMG)

Mandato 2011-2015

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

Suplentes - 2013-2015

Alessandro Fabrício Garcia (PUC-Rio)
Aline Maria Santos Andrade (UFBA)
Daltro José Nunes (UFRGS)
Karin Koogan Breitman (PUC-Rio)
Rodolfo Jardim de Azevedo (UNICAMP-IC)

Promoção

Laboratório Nacional de Redes de Computadores (LARC)

Diretoria 2012-2014

Diretor do Conselho Técnico-Científico

Elias P. Duarte Jr. (UFPR)

Diretor Executivo

Luciano Paschoal Gaspary (UFRGS)

Vice-Diretora do Conselho Técnico-Científico

Rossana Maria de C. Andrade (UFC)

Vice-Diretor Executivo

Paulo André da Silva Gonçalves (UFPE)

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, CEFET-CE, UFRN, UFES, UFBA, UNIFACS, UECE, UFPR, UFPA, UFAM, UFABC, PUCPR, UFMS, UnB, PUC-RS, UNIRIO, UFS e UFU.

Realização

Comitê de Organização

Coordenação Geral

Joni da Silva Fraga (UFSC)

Frank Augusto Siqueira (UFSC)

Coordenação do WTF

Luciana Arantes (Université de Paris 6)

Coordenação de Workshops

Carlos André Guimarães Ferraz (UFPE)

Realização

Organização Local

Carlos Barros Montez (UFSC)

Edison Tadeu Lopes Melo (UFSC)

Guilherme Eliseu Rhoden (PoP-SC)

Leandro Becker (UFSC)

Mário A. R. Dantas (UFSC)

Michelle Wangham (Univali)

Ricardo Felipe Custódio (UFSC)

Roberto Willrich (UFSC)

Rodrigo Pescador (PoP-SC)

Rômulo Silva de Oliveira (UFSC)

Secretaria do SBRC 2014

Juliana Clasen (UFSC)

Jade Zart (UFSC)

Mensagem do Coordenador de Workshops do SBRC 2014

Confirmando a consolidação nos últimos anos, este ano o Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC 2014) apresenta mais uma série de workshops, visando a discussão de temas novos e/ou específicos, como Internet do Futuro e Tolerância a Falhas. Os workshops envolvem comunidades focadas e oferecem oportunidades para discussões mais profundas e ampliação de conhecimentos, envolvendo pesquisadores e muitos estudantes em fase de desenvolvimento de seus trabalhos em andamento. Neste ano tivemos novas submissões, além dos workshops já considerados tradicionais parceiros do SBRC, o que representa o dinamismo da comunidade de Redes de Computadores e Sistemas Distribuídos no Brasil. Infelizmente, estas novas submissões não puderam ainda ser acomodadas, mas certamente serão consideradas para próximas edições do SBRC.

Neste SBRC 2014, temos a realização de workshops já consolidados no 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 de Computação em Clouds e Aplicações), o WP2P+ (Workshop de Redes P2P, Dinâmicas, Sociais e Orientadas a Conteúdo), o WRA (Workshop de Redes de Acesso em Banda Larga), o WoCCES (Workshop of Communication in Critical Embedded Systems), o WoSiDA (Workshop on Autonomic Distributed Systems) e o WPEIF (Workshop de Pesquisa Experimental da Internet do Futuro). Há que se mencionar a importante parceria com o WRNP (Workshop da Rede Nacional de Ensino e Pesquisa), que em sua 15a edição, cumpre o importante papel de fazer a ponte entre as comunidades técnica e científica da área. Não tenho dúvida que a qualidade técnica e científica dos workshops se manterá em alta compatível com o SBRC.

Agradeço aos Coordenadores Gerais, Joni da Silva Fraga e Frank Siqueira (UFSC), pelo convite para coordenar os workshops do SBRC 2014 e por todo o apoio recebido.

Desejo muito sucesso e excelente participação nos Workshops do SBRC 2014!

Carlos André Guimarães Ferraz (UFPE)

Coordenador de Workshops do SBRC 2014

Mensagem da Coordenadora do WTF 2014

Sejam bem vindos ao XV Workshop de Testes e Tolerância a Falhas (WTF), um evento que ocorre anualmente, promovido pela Comissão Especial de Sistemas Tolerantes a Falhas (CE-TF) da Sociedade Brasileira de Computação (SBC). A sua décima quinta edição será realizada no dia 5 de maio de 2014, Florianópolis, em conjunto com o XXXII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC).

O WTF é um fórum de debates e apresentações de trabalhos de pesquisas relacionados à confiança no funcionamento de sistemas, em especial nas áreas de testes, tolerância e detecção de falhas, dependabilidade, robutez e intrusões. Nesta edição, o WTF contemplará atividades de apresentação de artigos, além de uma palestra convidada, com o objetivo de suscitar discussões e interações frutuosas entre os participantes do workshop.

Todos os 19 artigos submetidos ao XV WTF tiveram 3 revisões, realizadas por um Comitê de Avaliação criterioso composto por 30 pesquisadores especialistas nas diversas áreas de interesse do workshop. Dentre estes, 13 foram selecionados para publicação e suas apresentações foram agrupadas em quatro sessões técnicas: Testes - Modelo, Automação, Estratégias e Avaliação (4 artigos); Algoritmos Distribuídos (5 artigos); e Tolerância a Falhas, Disponibilidade, Segurança e DTN (4 artigos).

Em complemento às sessões técnicas, convidamos Fernando Pedone, professor e pesquisador da Universidade de Lugano, Faculdade de Informática, que apresentará a palestra Geo-replicated Storage with Scalable Deferred Update Replication.

Gostaria de agradecer à CE-TF pelo convite para coordenar o XV WTF e principalmente a todos os participantes do Comitê de Programa e Revisores pelo excelente trabalho de avaliação, sugestões, reatividade e boas discussões, que contribuíram para termos uma seleção criteriosa, construtiva e dentro dos prazos estabelecidos. Agradeço também aos autores e Fernando Pedone, palestrante convidado, pelas suas valiosas contribuições assim como a Frank Siqueira e a Joni Fraga, organizadores do SBRC 2014, e a Carlos André Guimarães Ferraz, coordenador de Workshops, pelo apoio logístico para a concretização do XV WTF. Finalmente, um agradecimento para Fabíola Greve, pela coordenação das revisões anônimas, bem como a Eduardo Alchieri, coordenador do WTF 2013, e a Ricardo Neisse, administrador do JEMS, pelo suporte e ajuda.

Luciana Arantes

Coordenadora do WTF 2014

Comitê de Programa do WTF 2014

Alcides Calsavara (PUCPR)
Allyson Bessani (Universidade de Lisboa)
Ana Ambrosio (Instituto Nacional de Pesquisas Espaciais)
Avelino Zorzo (PUCRS)
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)
Jose Pereira (INESC TEC & U. Minho)
Lau Cheuk Lung (UFSC)
Luiz Eduardo Buzato (UNICAMP)
Marco Vieira (University of Coimbra)
Michele Nogueira (UFPR)
Miguel Correia (Instituto Superior Técnico)
Raimundo Barreto (UFAM)
Raul Ceretta Nunes (UFSM)
Regina Moraes (UNICAMP)
Rogerio de Lemos (University of Kent)
Rui Oliveira (Universidade do Minho)
Sergio Gorender (UFBA)
Taisy Weber (UFRGS)
Udo Fritzke Jr. (PUC Minas)

Sumário

Sessão Técnica 1 – Testes - Modelo, Automação, Estratégias e Avaliação.....1

Multi-Objective Test Case Selection: A study of the influence of the Catfish effect on PSO based strategies

Luciano Souza (IFNMG), Ricardo Prudêncio (UFPE) e
Flavia Barros (UFPE).....3

Uma Abordagem de Análise do Tempo de Resposta para Teste de Desempenho em Aplicações Web

Priscila Guarienti, Maicon Bernardino, Avelino Zorzo e
Flávio Moreira de Oliveira (PUCRS).....17

Uma Estratégia de Testes Logarítmica para o Algoritmo Hi-ADSD

Elias Duarte Jr., Luis Carlos De Bona e
Vinicius Kwiecien Ruoso (UFPR).....31

A Test Process Model to Evaluate Performance Impact of Privacy Protection Solutions

Victor Mello, Tania Basso e Regina Moraes (UNICAMP).....45

Sessão Técnica 2 – Algoritmos Distribuídos.....59

Distributed Snapshot algorithm for multi-active object-based applications

Michel Souza (UFBA) e Françoise Baude (Université de Nice Sophia
Antipolis - France).....61

Uma Solução Autonômica para a Criação de Quóruns Majoritários Baseada no VCube

Luiz Rodrigues (UNIOESTE/UFPR), Elias Duarte Jr. (UFPR) e
Luciana Arantes (Université de Paris VI).....74

Potência do Sinal de Recepção como Suporte à Detecção de Mobilidade em Detectores de Defeitos

Miguel Angelo Baggio, Raul Ceretta Nunes, Marcia Pasin e
Antonio Rodrigo Delepiane de Vit (UFSM).....88

Proposta de Implementação de Memória Compartilhada Dinâmica Tolerante a Falhas

Mateus Braga e Eduardo Alchieri (UnB).....102

Sumário

Serviço de Líder em Sistemas Dinâmicos com Memória Compartilhada

Catia Khouri (UESB) e Fabíola Greve (UFBA)..... 116

Sessão Técnica 3 – Tolerância a Falhas, Disponibilidade, Segurança e DTN.....131

Uma Avaliação de Abordagens de Distribuição para Gerenciamento de Redes Tolerantes a Atrasos e Desconexões

Ewerton Salvador (UFMG), Vinicius Mota (UFMG), Virgilio Almeida (UFMG), Daniel F. Macedo (FMG), José Marcos Nogueira (UFMG), Jéferson Nobre (UFRGS), Pedro A. Duarte (UFRGS) e Lisandro Z. Granville (UFRGS).....133

Aplicação da IEC 61508 na Prototipação de Protocolos Seguros de Comunicação

William Vidal, Rodrigo Dobler, Sérgio Luis Cechin, Taisy Weber e João Netto (UFRGS)..... 147

Uma Análise da Sobrecarga Imposta pelo Mecanismo de Replicação de Máquinas Virtuais Remus

Marcelo da Silva, Guilherme Koslovski e Rafael Obelheiro (UDESC)..... 160

Provedores de Identidade Resilientes e Confiáveis

Diego Kreutz (Faculty of Science of University of Lisbon - Portugal), Eduardo Feitosa (UFAM) e Hugo Cunha (UFAM)..... 174

Índice por Autor..... 189

32º Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos

Florianópolis - SC

XV Workshop de Testes e Tolerância a Falhas (WTF 2014)

Sessão Técnica 1

Testes - Modelo, Automação, Estratégias e Avaliação

Multi-Objective Test Case Selection: A study of the influence of the Catfish effect on PSO based strategies

Luciano S. de Souza[1,2], **Ricardo B. C. Prudêncio**[2], **Flavia de A. Barros**[2]

[1]Federal Institute of Education Science and Technology of the North of Minas Gerais
IFNMG – Pirapora (MG), Brazil

[2]Center of Informatics (CIn)
Federal University of Pernambuco (UFPE)
Recife (PE), Brazil

luciano.souza@ifnmg.edu.br, {rbcp,fab}@cin.ufpe.br

Abstract. *During the software testing process many test suites can be generated in order to evaluate and assure the quality of the products. In some cases, the execution of all suites can not fit the available resources (time, people, etc). Hence, automatic Test Case (TC) selection could be used to reduce the suites based on some selection criterion. This process can be treated as an optimization problem, aiming to find a subset of TCs which optimizes one or more objective functions (i.e., selection criteria). In this light, we developed mechanisms for TC selection in context of structural and functional testing. The proposed algorithms consider two objectives simultaneously: maximize branch coverage (or functional requirements coverage) while minimizing execution cost (time). These mechanisms were implemented by deploying multi-objective techniques based on Particle Swarm Optimization (PSO). Additionally, we added the so-called catfish effect into the multi-objective selection algorithms in order to improve their results. The performed experiments revealed the feasibility of the proposed strategies.*

1. Introduction

During the software development process, the software testing activities has grown in importance due to the need of high quality products. Software testing activities aim to assure quality and reliability of the developed products. Nevertheless, this activity is sometimes neglected, since it is very expensive, reaching up to 40% of the final software development cost [Ramler and Wolfmaier 2006]. In this scenario, automation seems to be the key solution for improving the efficiency and effectiveness of the software testing process.

The related literature presents two main approaches for testing: White Box (structural) or Black Box (functional) testing. Structural testing investigates the behavior of the software directly accessing its code. Functional testing, in turn, investigates whether the software functionalities are responding/behaving as expected without using any knowledge about the code [Young and Pezze 2005]. In both approaches, the testing process relies on the (manual or automatic) generation and execution of a Test Suite (TS), that is, a set of Test Cases. A Test Case (TC), consists of "a set of inputs, execution conditions, and a pass/fail conditions" [Young and Pezze 2005]. The execution of the TS aims

to provide a suitable coverage of an adopted test adequacy criterion (e.g., code coverage, functional requirements coverage) in order to satisfy the test goals. In this scenario, both automatic generated or manually created TS may be large. In general, they try to cover all possible test scenarios in order to accomplish a good coverage of the adequacy criterion. Although it is desirable to fully satisfy the test goals, the execution of large suites is a very expensive task, demanding a great deal of the available resources (time and people) [Harold et al. 1993].

It is possible identify that large test suites contain some redundancies (i.e., two or more TCs covering the same requirement/piece of code). Hence, it is possible to reduce the suite in order to fit the available resources. This task of reducing a test suite based on a given selection criterion is known as *Test Case selection* [Borba et al. 2010]. TC selection is not easy or trivial since there may be a large number of TC combinations to consider when searching for an adequate TC subset.

Clearly, TC selection should not be performed at random, in order to preserve the coverage of the testing criterion. In the absence of automatic tools, this task is usually manually performed in an *ad-hoc* fashion. However, manual TC selection is time-consuming and susceptible to errors. Thus, the related literature focus on deploying automatic tools for TC selection.

A promising approach for the TC selection problem is to treat it as a search based optimization problem [Harman 2011]. This waym the search techniques explore the space of possible solutions searching for a TC subset in order to optimize a given objective function, i.e., the given selection criterion (see [Ma et al. 2005], [Yoo and Harman 2007], [Barltrop et al. 2010], [de Souza et al. 2010], [de Souza et al. 2013]). In most of the testing environments, there is not only a single selection criterion to be considered. In fact, it is very common the need to take into account, during the selection process, more than one criterion. Following this light, we can cite the use of multi-objective evolutionary approaches [Yoo and Harman 2007, Yoo and Harman 2010b, Yoo et al. 2011b, Yoo et al. 2011a] and the use of multi-objective Particle Swarm Optimization (PSO) techniques [de Souza et al. 2011] (our previous work).

In [de Souza et al. 2011], we investigated the multi-objective TC selection considering both the functional requirements coverage (quality), and the execution effort (cost) of the selected subset of TCs as objectives of the selection process. There, we developed the Binary Multi-Objective PSO techniques for TC selection by combining: (1) the binary version of PSO proposed in [Kennedy and Eberhart 1997]; and (2) the multi-objective PSO algorithms MOPSO [Coello et al. 2004] and MOPSO-CDR [Santana et al. 2009]. Despite the good results of the implemented techniques (BMOPSO-CDR and BMOPSO) on a case study, further improvements could be performed.

The use of the so-called "catfish" effect, for instance, has shown to improve the performance of the single objective version of the binary PSO [Chuang et al. 2011]. Hence, in the current work, we investigate whether this catfish effect can improve the performance of our former algorithms BMOPSO-CDR and BMOPSO [de Souza et al. 2011]. For that, we created two new multi-objective strategies, the CatfishBMOPSO-CDR and CatfishBMOPSO, by adding the catfish effect. We point out that the use of catfish effect, to the best of our knowledge, was not yet investigated in the

context of test case selection or in the context of multi-objective optimization.

Additionally, in order to measure the performance of the proposed strategies, we performed experiments using two testing scenarios (structural and functional testing). For structural testing, we used test suites from five programs from the Software-artifact Infrastructure Repository (SIR) [Do et al. 2005], and for the functional testing, we used two unrelated test suites from the context of mobile devices. Each implemented algorithm returns to the user a set of solutions (test suites) with different values of branch coverage (for structural testing) or functional requirements coverage (for functional testing) *versus* execution cost. The user (tester) can then choose the solution that best fits into the available resources. It is important to highlight that, although the focus of our research is the multi-objective TC selection problem, the proposed techniques can also be applied to binary multi-objective optimization in other contexts.

Finally, it is also important to highlight that, to the best of our knowledge, there is no other work, in literature, that applied PSO techniques into the multi-objective TC selection problem (besides our previous work [de Souza et al. 2011]). Furthermore, we point out that no other multi-objective TC selection work performed both structural and functional testing in the same work. Hence, this is an promising study area, and this work aims to explore it a little further by creating new multi-objective strategies for test case selection.

The next section briefly presents search based concepts for TC selection. Section 3 explains our approach of multi-objective PSO for TC selection. Sections 3.2, 3.3 and 3.4 detail the implemented selection algorithms. Section 4 details the experiments performed to evaluate the proposed algorithms and presents the obtained results. Finally, Section 5 presents some conclusions and future work.

2. Search Based Test Case Selection

As discussed before, although testing is central in the software development process, it is sometimes neglected due to its high cost. A primary way to reduce software testing costs is by reducing the size of test suites, since TS execution is resource consuming, and this approach is known as Test Case selection. Given an input TS, TC selection aims to find a relevant TC subset regarding some test adequacy criterion (such as the amount of code or functional requirements covered by the chosen TC subset, for instance). Clearly, this task should not be performed at random, since a random choice seldom returns a representative TC subset.

A very promising approach for TC selection is to treat this task as a search optimization problem [Yoo and Harman 2010a]. In this approach, search techniques explore the space of possible solutions (subsets of TC), seeking the solution that best matches the test objectives.

When analyzing which search based approach to use, we initially disregard the exhaustive (brute-force) search techniques, since we are facing an NP-complete problem [Lin and Huang 2009]. We also disregard random search since, when dealing with large search spaces, random choices seldom deliver a representative TC subset regarding the adopted test adequacy criterion. In this scenario, more sophisticated approaches, as search based optimization techniques, should be considered to treat this problem.

As known, optimization techniques in general demand the use of one or more objective (fitness) functions, which will determine the quality of each possible solution (a TC subset) regarding some chosen search criterion. Each fitness function corresponds to a different search objective. Regarding single-objective search techniques applied to test case selection, we highlight the use of Simulated Annealing [Mansour and El-Fakih 1999], conventional Genetic Algorithms [Ma et al. 2005], [Mansour and El-Fakih 1999] and PSO [de Souza et al. 2010]. Regarding multi-objective techniques, we highlight [Yoo and Harman 2007, Yoo and Harman 2010b, Yoo et al. 2011b, Yoo et al. 2011a, de Souza et al. 2011].

In this work, we developed two new multi-objective algorithms (the CatfishBMOPSO-CDR and CatfishBMOPSO) by adding the catfish effect into the BMOPSO-CDR and BMOPSO (see [de Souza et al. 2011]). Furthermore, the proposed algorithms were applied to select structural and functional test suites. The objective functions to be optimized were the branch coverage (for structural testing) or functional requirements coverage (for functional testing), and the execution cost (in time) of the selected TCs (for both approaches), in such a way that we maximize the first function (coverage) and minimize the second one (cost). The proposed algorithms return a set of non-dominated solutions (a Pareto frontier) considering the aforementioned objectives. By receiving a set of diverse solutions, the user can choose the best one taking into account its current goals and available resources (e.g., the amount of time available at the moment to execute the test cases).

It is not the purpose of this work to enter into a discussion concerning which objectives are more important for the TC selection problem. Irrespective of arguments about their suitability, branch or functional requirements coverage are likely candidates for assessing quality of a TS, and execution time is one realistic measure of cost.

Furthermore, we can point out some previous work that applied PSO in Software Testing, particularly for test case generation [Windisch et al. 2007] and for regression testing prioritization [Kaur and Bhatt 2011]. We can observe that the literature lacks works applying multi-objective PSO into the TC selection problem. For functional testing selection, we can only can only point out our previous work (see [de Souza et al. 2011]), and for structural testing selection, to the best of our knowledge, there is no previous work. Moreover, the catfish effect was not yet applied into a multi-objective context or into TC selection problem. Hence, this is an promising study area and we explore it a little further in this work by deploying new strategies that take this into account.

Following, Section 3 presents details of our multi-objective approach for test case selection, and Section 4 brings the experiments performed to evaluate the proposed algorithms.

3. Multi-Objective PSO to Test Case Selection

In this work, we propose some Particle Swarm Optimization (PSO) based algorithms to solve multi-objective TC selection problems. In contrast to single-objective problems, Multi-Objective Optimization (MOO) aims to optimize more than one objective at the same time.

A MOO problem considers a set of k objective functions $f_1(x), f_2(x), ..., f_k(x)$ in which x is an individual solution for the problem being solved. The output of a MOO

algorithm is usually a population of non-dominated solutions considering the objective functions. Formally, let x and x' be two different solutions. We say that x dominates x' (denoted by $x \preceq x'$) if x is better than x' in at least one objective function and x is not worse than x' in any objective function. x is said to be *not dominated* if there is no other solution x_i in the current population, such that $x_i \preceq x$. The set of non-dominated solutions in the objective space returned by a MOO algorithm is known as Pareto frontier.

The PSO algorithm is a population-based search approach, inspired by the behavior of birds flocks [Kennedy and Eberhart 1995] and has shown to be a simple and efficient algorithm compared to other search techniques, including for instance the widespread Genetic Algorithms [Eberhart and Shi 1998]. The basic PSO algorithm starts its search process with a random population (also called swarm) of *particles*. Each particle represents a candidate solution for the problem being solved and it has four main attributes:

1. the position (**t**) in the search space (each position represents an individual solution for the optimization problem);
2. the current velocity (**v**), indicating a direction of movement in the search space;
3. the best position ($\hat{\mathbf{t}}$) found by the particle (the memory of the particle);
4. the best position ($\hat{\mathbf{g}}$) found by the particle's neighborhood (the social guide of the particle).

For a number of iterations, the particles fly through the search space, being influenced by their own experience $\hat{\mathbf{t}}$ and by the experience of their neighbors $\hat{\mathbf{g}}$. Particles change position and velocity continuously, aiming to reach better positions and to improve the considered objective functions.

3.1. Problem Formulation

In this work, the particle's positions were defined as binary vectors representing candidate subsets of TCs to be applied in the software testing process. Let $T = \{T_1, \ldots, T_n\}$ be a test suite with n test cases. A particle's position is defined as $\mathbf{t} = (t_1, \ldots, t_n)$, in which $t_j \in \{0, 1\}$ indicates the presence (1) or absence (0) of the test case T_j within the subset of selected TCs.

As said, two objective functions were adopted. The coverage (code branches or functional requirements) objective (to be maximized) represents the amount (in percentage) of code branches (or functional requirements) covered by a solution **t** in comparison to the amount of covered by T. Formally, let $C = \{C_1, \ldots, C_k\}$ be a given set of k branches/functional requirements covered by the original suite T. Let $F(T_j)$ be a function that returns the subset of branches/functional requirements in C covered by the individual test case T_j. Then, the coverage of a solution **t** is given as:

$$C_Coverage(\mathbf{t}) = 100 * \frac{|\bigcup_{t_j=1}\{F(T_j)\}|}{k} \tag{1}$$

In eq. (1), $\bigcup_{t_j=1}\{F(T_j)\}$ is the union of branches/functional requirements subsets covered by the selected test cases (i.e., T_j for which $t_j = 1$).

The other objective function (to be minimized) is the execution cost (the amount of time required to execute the selected suite). Formally, each test case $T_j \in T$ has a *cost score* c_j. The total cost of a solution **t** is then defined as:

$$Cost(\mathbf{t}) = \sum_{t_j=1} c_j \tag{2}$$

Finally, the MOO algorithms will be used to find a good Pareto frontier regarding the objective functions $C_Coverage$ and $Cost$.

3.2. The BMOPSO-CDR algorithm

The Binary Multi-objective Particle Swarm Optimization with Crowding Distance and Roulette Wheel (BMOPSO-CDR) was firstly presented in [de Souza et al. 2011]. It uses an External Archive (EA) to store the non-dominated solutions found by the particles during the search process. See [de Souza et al. 2011] for more details of BMOPSO-CDR algorithm.

The following summarizes the BMOPSO-CDR:

1. Randomly initialize the swarm, evaluate each particle according to the considered objective functions and then store in the EA the particles' positions that are non-dominated solutions;
2. WHILE stop criterion is not verified DO
   (a) Compute the velocity **v** of each particle as:

   $$\mathbf{v} \leftarrow \omega\mathbf{v} + C_1 r_1(\hat{\mathbf{t}} - \mathbf{t}) + C_2 r_2(\hat{\mathbf{g}} - \mathbf{t}) \tag{3}$$

   where ω represents the inertia factor; r_1 and r_2 are random values in the interval [0,1]; C_1 and C_2 are constants. The social guide ($\hat{\mathbf{g}}$) is defined as one of the non-dominated solutions stored in the current EA.

   (b) Compute the new position **t** of each particle for each dimension t_j as:

   $$t_j = \begin{cases} 1, & \text{if } r_3 \leq sig(v_j) \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

   where r_3 is a random number sampled in the interval [0,1] and sig(v_j) is defined as:

   $$sig(v_j) = \frac{1}{1 + e^{-v_j}} \tag{5}$$

   (c) Use the mutation operator as proposed by [Coello et al. 2004];
   (d) Evaluate each particle of the swarm and update the solutions stored in the EA;
   (e) Update the particle's memory $\hat{\mathbf{t}}$;
3. END WHILE and return the current EA as the Pareto frontier.

3.3. The BMOPSO algorithm

The BMOPSO algorithm was created by merging the Multi-Objective Particle Swarm Optimization proposed by Coello Coello *et. al* [Coello et al. 2004] and the binary version of PSO proposed by Kennedy and Eberhart [Kennedy and Eberhart 1997].

Instead of using of using the crowding distance concept on EA's solutions, it generates hypercubes on EA's objective space and locate each solution within those hypercubes. Each hypercube has a fitness corresponding to the number of solutions located in

it. Hence, when choosing the social guide $\hat{\mathbf{g}}$, a Roulette Wheel is used in order to favor the choose of a solution in a less crowded hypercube. For more details about the BMOPSO algorithm see [de Souza et al. 2011].

3.4. The CatfishBMOPSO-CDR and CatfishBMOPSO algorithms

The catfish effect derives its name from an effect that Norwegian fishermen observed when they introduced catfish into a holding tank for caught sardines. The introduction of a catfish, which is different from sardines, into the tank resulted in the stimulation of sardine movement, thus keeping the sardines alive and therefore fresh for a longer time [Chuang et al. 2011].

Similarly, the catfish effect is applied to the PSO algorithm in a such way that we introduce "catfish particles" in order to stimulate a renewed search by the rest of the swarm's particles (the sardines). Hence, these catfish particles helps to guide the particles, which can be trapped in a local optimum, to new regions of the search space, and thus leading to possible better areas [Chuang et al. 2011].

Chuang *et al.* [Chuang et al. 2011] proposed that, if the best value found by the swarm does not change after some iterations, it is considered stuck in a local optimum. Under such circumstances, 10% of particles with the worst fitness value are removed and substituted by catfish particles. The catfish particles are randomly positioned at extreme points of the search space, and stimulate a renewed search process for the particles of the swarm.

As the BMOPSO-CDR is an multi-objective algorithm, we needed to adapt the catfish effect in order to create the CatfishBMOPSO-CDR. For that, the following steps were introduced into the main loop (2) after step (e) of the BMOPSO-CDR algorithm:

1. IF the EA is not updated for 24 consecutive iterations[1] THEN
2. FOR the number of times equivalent of 10% of swarm size DO
   (a) Create a new solution $\mathbf{t}$
   (b) IF $r_4 < 0.5$ THEN
   $t_j = 1, \quad \forall t_j \in \mathbf{t}$
   (c) ELSE
   $t_j = 0, \quad \forall t_j \in \mathbf{t}$
   where r_4 is a random number sampled in the interval [0,1]
3. Randomly replace 10% of swarm's solutions by the catfish solutions.

In the single objective approach for the catfish effect (see [Chuang et al. 2011]), the authors needed to compare all the current solutions in order to find the worst solutions (particles) to be replaced by the new ones (catfish particles). Contrarily, as in our MOO approach we use an EA to store all best solutions (non-dominated solutions), we decided to keep the addition of the catfish effect as simple as possible by randomly choosing particles of current swarm to be replaced, instead of making comparisons to find the worst ones.

Additionally, the CatfishBMOPSO algorithm was created by adding the catfish effect into the BMOPSO algorithm using the same process of the CatfishBMOPSO-CDR.

[1] In [Chuang et al. 2011] they verify if the best value was not updated for 3 consecutive iterations, but, after preliminary trial and error parameter tuning, we discover that 24 is a better value for our context. Nevertheless, some future work is necessary in order to find the best value for our context.

4. Experiments and Results

This section presents the experiments performed in order to evaluate the search algorithms implemented in this work. The proposed multi-objective catfish approaches were compared whit their former algorithms in order to verify wether the addition of catfish effect has impact in the performance. For the structural testing scenario, the experiments were performed using 5 programs from SIR [Do et al. 2005]. In turn, the functional testing scenario, used test suites from the context of mobile devices[2].

4.1. Experiments Preparation

For the structural testing selection, initially we selected 5 programs from SIR: part of the *Siemens* suite (printtokens, printtokens2, schedule, schedule2), and the program space from the European Space Agency. These real world applications range from 374 to 6,199 lines of code. Since each program has a large number of available test suites we randomly selected 4 suites (referred here as T_1, T_2, T_3 and T_4) for each program. In Table 1 we give more information about the programs.

Table 1. Characteristics of the SIR programs

Program	Lines of Code	Avg. suite size
printtokens	726	16
printtokens2	570	17
schedule	412	10
schedule2	374	12
space	6199	157

The execution cost information was computed for each test case by using the Valgrind profiling tool [Nethercote and Seward 2003], as proposed in [Yoo and Harman 2007]. The execution time of test cases is hard to measure accurately due to the fact that it involves many external parameters that can affect the execution time as different hardware, application software and operating system. In order to circumvent these issues, we used Valgrind, which executes the program binary code in an emulated, virtual CPU. The computational cost of each test case was measured by counting the number of virtual instruction codes executed by the emulated environment and it allows to argue that these counts are directly proportional to the cost of the test case execution. Additionally, the branch coverage information was measured using the profiling tool **gcov** from the GNU compiler **gcc** (also proposed in [Yoo and Harman 2007]).

On the other hand, for the functional testing selection, we selected 2 test suites related to mobile devices: an Integration Suite (IS) and a Regression Suite (RS). Both suites have 80 TCs, each one representing an functional testing scenario. The IS is focused on testing whether the various features of the mobile device can work together, i.e., whether the integration of the features behaves as expected. The RS, in turn, is aimed at testing whether updates to a specific main feature (e.g., the message feature) have not introduced faults into the already developed (and previously tested) feature functionalities. Contrarily to the structural suites, where each suite is intended to testing the related

[2]These suites were created by test engineers of the Motorola CIn-BTC (Brazil Test Center) research project.

program almost as whole, the used functional suites are related to a much more complex environment. Hence, just a little portion of the mobile device operational system is tested.

The cost to execute each test case of the functional suite was measured by the *Test Execution Effort Estimation Tool*, developed by [Aranha and Borba 2008]. The effort represents the cost (in time) needed to manually execute each test case on a particular mobile device. Each TC has annotated which requirements it covers, thus we used this information in order to calculate the functional requirements coverage.

4.2. Metrics

In our experiments, we evaluated the results (i.e., the Pareto frontiers) obtained by the algorithms, for each test suite, according to four different quality metrics usually adopted in the literature of multi-objective optimization. The following metrics were adopted in this paper: Hypervolume (HV) [Deb and Kalyanmoy 2001], Generational Distance (GD) [Coello et al. 2007], Inverted Generational Distance (IGD) [Coello et al. 2007] and Coverage (C) [Deb and Kalyanmoy 2001]. Each metric considers a different aspect of the Pareto frontier.

1. Hypervolume (HV) [Deb and Kalyanmoy 2001]: computes the size of the dominated space, which is also called the *area under the curve*. A high value of hypervolume is desired in MOO problems.
2. Generational Distance (GD) [Coello et al. 2007]: The Generational Distance (GD) reports how far, on average, one pareto set (called PF_{known}) is from the true pareto set (called as PF_{true}).
3. Inverted Generational Distance (IGD) [Coello et al. 2007]: is the inverse of GD by measuring the distance from the PF_{true} to the PF_{known}. This metric is complementary to the GD and aims to reduce the problem when PF_{known} has very few points, but they all are clustered together. So this metric is affected by the distribution of the solutions of PF_{known} comparatively to PF_{true}.
4. Coverage (C) [Deb and Kalyanmoy 2001]: The Coverage metric indicates the amount of the solutions within the non-dominated set of the first algorithm which dominates the solutions within the non-dominated set of the second algorithm.

Both GD and IGD metrics requires that the PF_{true} be known. Unfortunately, for more complex problems (with bigger search spaces), as the *space* program, it is impossible to know PF_{true} a priori. In these cases, instead, a reference Pareto frontier (called here $PF_{reference}$) can be constructed and used to compare algorithms regarding the Pareto frontiers they produce [Yoo and Harman 2010b]. The reference frontier represents the union of all found Pareto frontiers, resulting in a set of non-dominated solutions found. Additionally, the C metric reported in this work refers to the coverage of the optimal set $PF_{reference}$, over each algorithm, indicating the amount of solutions of those algorithms that are dominated, e. g. that are not optimal.

4.3. Algorithms Settings

All the algorithms are run for a total of 200,000 objective function evaluations. The algorithms used 20 particles, ω linearly decreasing from 0.9 to 0.4, constants C_1 and C_2 of 1.49, maximum velocity of 4.0 and EA's size of 200 solutions. These values were the same used in [de Souza et al. 2011] and represent generally used values in the literature. As previously said, the catfish effect is applied whenever the EA not changes for 24 consecutive iterations.

4.4. Results

After 30 executions of each TC selection algorithm, the values of branch/functional requirements coverage and execution cost observed in the Pareto frontiers were normalized, since they are measured using different scales. All the evaluation metric were computed and statistically compared using the *Mann-Whitney U test*.

The *Mann-Whitney U test* is a nonparametric hypothesis test that does not require any assumption on the parametric distribution of the samples. It tests the null hypothesis that two independent groups are equal. In the context of this paper, the null hypothesis states that, regarding the observed metric, two different algorithms produce equivalent Pareto frontiers. The α level was set to 0.95, and significant *p-values* suggest that the null hypothesis should be rejected in favor of the alternative hypothesis, which states that the Paretor frontiers are different.

For the *Siemens* programs (printtokens, printtokens2, schedule and schedule2) we performed an exhaustive search and found the true optimal Pareto frontier. This frontier was compared with those produced by the selection algorithms, and we observed that, for all cases, the selection algorithms were capable to find the true Pareto frontier, thus the null hypothesis could not be rejected and, for the *Siemens* programs, all algorithms were statistically equivalent. This happened due to the fact of the search spaces of the *Siemens* programs were small, allowing the algorithms to easily find the optimal solution.

Additionally, the values of the metrics for the *Siemens* programs were not presented here because, as the values were equal, no important information can be retrieved from them. Contrarily, the space program has a bigger search space (an average of 2^{157} combinations) and the algorithms performed differently in some cases. Furthermore, for both space and mobile suites we do not performed exhaustive search due to the high computational cost.

Tables 2 and 3 present the average values of the adopted metrics (for the space and mobile suites), as well as the observed standard deviations. Additionally, asterisks (*) were used to indicate whenever two values are considered statistically equal (i.e. the null hypothesis was not rejected).

Regarding HV, IGD and C metrics, we can observe that the CatfishBMOPSO-CDR and CatfishBMOPSO outperformed the others. This means that: they dominated bigger objective space areas; were better distributed comparatively to optimal Pareto set (represented by the $PF_{reference}$); and were less dominated by the optimal Pareto set. However, regarding the GD metric, the catfish approaches worsened the former algorithms. It is possible to observe that, for the GD metric, the BMOPSO-CDR and BMOPSO outperformed the catfish approaches in almost all cases (except for the mobile regression suite where the CatfishBMOPSO was not outperformed). This means, that the former algorithms produce Pareto frontiers with better convergence to the optimal Pareto set.

Performing a deeper analysis on the produced Pareto frontiers, we observed that the frontiers produced by CatfishBMOPSO-CDR and CatfishBMOPSO had more solutions and were bigger than the frontiers produced by BMOPSO-CDR and BMOPSO. This happens because, when introducing the catfish particles into the extremities of the search space, it is expected that the catfish algorithms can find more solutions as well as find solutions near to the extremities. However, by introducing the catfish particles, the

Table 2. Mean value and standard deviation for the metrics - Program space

Metric	Suite	BMOPSO-CDR	BMOPSO	CatfishBMOPSO-CDR	CatfishBMOPSO
HV	T_1	0.804 (0.008)	0.799 (0.009)	0.864* (0.010)	0.861* (0.008)
	T_2	0.779* (0.007)	0.776* (0.007)	0.847** (0.006)	0.844** (0.010)
	T_3	0.815* (0.007)	0.812* (0.007)	0.871 (0.008)	0.864 (0.009)
	T_4	0.795 (0.007)	0.789 (0.008)	0.857 (0.008)	0.849 (0.008)
GD	T_1	0.005* (0.001)	0.004* (0.001)	0.012** (0.002)	0.013** (0.001)
	T_2	0.004* (8.8E-4)	0.004* (8.5E-4)	0.005 (8.8E-4)	0.004 (9.8E-4)
	T_3	0.004* (0.001)	0.004* (9.2E-4)	0.005 (9.2E-4)	0.004* (9.2E-4)
	T_4	0.003* (6.7E-4)	0.003* (5.9E-4)	0.003** (8.2E-4)	0.003** (8.0E-4)
IGD	T_1	0.030* (0.001)	0.030* (0.001)	0.004 (6.2E-4)	0.005 (0.002)
	T_2	0.027* (0.001)	0.027* (0.001)	0.003 (7.4E-4)	0.004 (7.4E-4)
	T_3	0.027* (0.001)	0.027* (0.001)	0.003 (5.7E-4)	0.004 (7.0E-4)
	T_4	0.026 (0.001)	0.028 (0.001)	0.003 (5.0E-4)	0.003 (6.8E-4)
C	T_1	0.99* (0.01)	0.97* (0.07)	0.94** (0.03)	0.95** (0.01)
	T_2	0.97* (0.04)	0.98* (0.03)	0.92** (0.02)	0.92** (0.01)
	T_3	0.97 (0.04)	0.98 (0.08)	0.91* (0.03)	0.91* (0.02)
	T_4	0.98* (0.02)	0.98* (0.02)	0.93** (0.02)	0.92** (0.02)

Table 3. Mean value and standard deviation for the metrics - Mobile suites

Metric	Suite	BMOPSO-CDR	BMOPSO	CatfishBMOPSO-CDR	CatfishBMOPSO
HV	IS	0.599 (0.005)	0.593 (0.005)	0.621* (0.003)	0.621* (0.004)
	RS	0.800 (0.008)	0.793 (0.007)	0.815* (0.005)	0.815* (0.006)
GD	IS	0.002* (3.2E-4)	0.002* (2.5E-4)	0.003 (2.4E-4)	0.002* (1.8E-4)
	RS	0.004* (6.1E-4)	0.004* (5.1E-4)	0.010** (0.002)	0.011** (0.001)
IGD	IS	0.007 (8.6E-4)	0.008 (8.9E-4)	0.002 (2.1E-4)	0.83 (0.01)
	RS	0.014 (0.002)	0.016 (0.002)	0.005* (0.001)	0.005* (0.001)
C	IS	0.98 (0.01)	0.97 (0.02)	0.96* (0.01)	0.95* (0.01)
	RS	0.98 (0.01)	0.97 (0.03)	0.96* (0.01)	0.95* (0.02)

algorithms seemed to lose some sort of capability to refine the already found solutions (the convergence was affected) and this directly impacted the GD metric.

Comparing the CatfishBMOPSO-CDR and CatfishBMOPSO, it is not possible to point out the best, since there is no clear domination between them. For most of the metrics they are statistically equal and sometimes one outperforms the other. In turn, the same can be said about the comparison between BMOPSO-CDR and BMOPSO. The results presented in [de Souza et al. 2011] differs from the presented here as there, the BMOPSO-CDR algorithm was never outperformed by the BMOPSO. This happened due to the use of different metrics to evaluate the algorithms. A future work is planned in order to address which metrics are the best for our context.

5. Conclusion

In this work, we propose the addition of the catfish effect into the binary multi-objective PSO approaches for both structural and functional TC selection. The main contribution of the current work was to investigate whether this effect can improve the multi-objective PSO approaches, from [de Souza et al. 2011], for selecting structural and functional test cases considering both branch/functional requirements coverage and execution cost. We highlight that the catfish effect in a multi-objective PSO was not yet investigated in the context of TC selection (or in any other context). We also point out that the developed selection algorithms can be adapted to other test selection criteria and are not limited to two objective functions. Furthermore, we expect that the good results can also be obtained on other application domains.

Expecting that the addition of catfish effect into multi-objective PSO could improve its performance, we implemented two catfish binary multi-objective PSO algorithms. We compared them with our former multi-objective PSO algorithms (the BMOPSO-CDR and BMOPSO). In the performed experiments, the CatfishBMOPSO-CDR and CatfishBMOPSO outperformed the others for all the metrics (except for the GD metric). Hence, we can conclude that the catfish effect indeed introduced improvements to the algorithms, but more research is necessary in order to prevent the negative impact to the algorithms' capacity of refine solutions. As future work we pretend to investigate the use of local search algorithms in order to deal with this drawback.

Another future work is to perform the same experiments on a higher number of programs in order to verify whether the obtained results are equivalent to those presented here and whether these results can be extrapolated to other testing scenarios other than the presented. Furthermore, we plan to determine whether the adopted metrics are indeed suitable to measure the quality of test suites.

Also, we will investigate the impact of changing the PSO's parameters in its performance on the TC selection problem. Besides, we intend to implement add the catfish effect with other multi-objective approaches for a more complete comparison between techniques and to determine the distinct advantages of using the BMOPSO-CDR and BMOPSO as basis.

References

Aranha, E. and Borba, P. (2008). Using process simulation to assess the test design effort reduction of a model-based testing approach. In *ICSP*, pages 282–293.

Barltrop, K., Clement, B., Horvath, G., and Lee, C.-Y. (2010). Automated test case selection for flight systems using genetic algorithms. In *Proceedings of the AIAA Infotech@Aerospace Conference (I@A 2010). Atlanta, GA*.

Borba, P., Cavalcanti, A., Sampaio, A., and Woodcock, J., editors (2010). *Testing Techniques in Software Engineering, Second Pernambuco Summer School on Software Engineering, PSSE 2007, Recife, Brazil, December 3-7, 2007, Revised Lectures*, volume 6153 of *Lecture Notes in Computer Science*. Springer.

Chuang, L.-Y., Tsai, S.-W., and Yang, C.-H. (2011). Improved binary particle swarm optimization using catfish effect for feature selection. *Expert Syst. Appl.*, 38(10):12699–12707.

Coello, C., Pulido, G., and Lechuga, M. (2004). Handling multiple objectives with particle swarm optimization. *IEEE Transactions on Evolutionary Computation*, 8(3):256–279.

Coello, C. A. C., Lamont, G. B., and van Veldhuizen, D. A. (2007). *Evolutionary Algorithms for Solving Multi-Objective Problems*, volume 5. Springer.

de Souza, L. S., Miranda, P. B. C., Prudêncio, R. B. C., and Barros, F. d. A. (2011). A multi-objective particle swarm optimization for test case selection based on functional requirements coverage and execution effort. In *In Proceedings of the 23rd International Conference on Tools with Artificial Intelligence (ICTAI 2011)*, Boca Raton, FL, USA.

de Souza, L. S., Prudêncio, R. B., de A. Barros, F., and da S. Aranha, E. H. (2013). Search based constrained test case selection using execution effort. *Expert Systems with Applications*, 40(12):4887 – 4896.

de Souza, L. S., Prudêncio, R. B. C., and Barros, F. d. A. (2010). A constrained particle swarm optimization approach for test case selection. In *In Proceedings of the 22nd International Conference on Software Engineering and Knowledge Engineering (SEKE 2010)*, Redwood City, CA, USA.

Deb, K. and Kalyanmoy, D. (2001). *Multi-Objective Optimization Using Evolutionary Algorithms*. Wiley, 1 edition.

Do, H., Elbaum, S., and Rothermel, G. (2005). Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 10(4):405–435.

Eberhart, R. C. and Shi, Y. (1998). Comparison between genetic algorithms and particle swarm optimization. *LNCS*, 1447:611–616.

Harman, M. (2011). Making the case for morto: Multi objective regression test optimization. In *Fourth International IEEE Conference on Software Testing, Verification and Validation*, pages 111–114. IEEE Computer Society.

Harold, M. J., Gupta, R., and Soffa, M. L. (1993). A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285.

Kaur, A. and Bhatt, D. (2011). Hybrid particle swarm optimization for regression testing. *International Journal on Computer Science and Engineering*, 3 (5):1815–1824.

Kennedy, J. and Eberhart, R. C. (1995). Particle swarm optimization. In *Proceedings of the IEEE International Joint Conference on Neural Networks*, pages 1942–1948.

Kennedy, J. and Eberhart, R. C. (1997). A discrete binary version of the particle swarm algorithm. In *Proceedings of the World Multiconference on Systemics, Cybernetics and Informatics*, pages 4104–4109.

Lin, J.-W. and Huang, C.-Y. (2009). Analysis of test suite reduction with enhanced tie-breaking techniques. *Inf. Softw. Technol.*, 51(4):679–690.

Ma, X.-Y., Sheng, B.-K., and Ye, C.-Q. (2005). Test-suite reduction using genetic algorithm. *Lecture Notes in Computer Science*, 3756:253–262.

Mansour, N. and El-Fakih, K. (1999). Simulated annealing and genetic algorithms for optimal regression testing. *Journal of Software Maintenance*, 11(1):19–34.

Nethercote, N. and Seward, J. (2003). Valgrind: A program supervision framework. In *In Third Workshop on Runtime Verification (RVï¿½03)*.

Ramler, R. and Wolfmaier, K. (2006). Economic perspectives in test automation - balancing automated and manual testing with opportunity cost. In *Workshop on Automation of Software Test, ICSE 2006*.

Santana, R. A., Pontes, M. R., and Bastos-Filho, C. J. A. (2009). A multiple objective particle swarm optimization approach using crowding distance and roulette wheel. In *Proceedings of the 2009 Ninth International Conference on Intelligent Systems Design and Applications*, ISDA '09, pages 237–242, Washington, DC, USA. IEEE Computer Society.

Windisch, A., Wappler, S., and Wegener, J. (2007). Applying particle swarm optimization to software testing. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, GECCO '07, pages 1121–1128, New York, NY, USA. ACM.

Yoo, S. and Harman, M. (2007). Pareto efficient multi-objective test case selection. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 140–150.

Yoo, S. and Harman, M. (2010a). Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120.

Yoo, S. and Harman, M. (2010b). Using hybrid algorithm for pareto efficient multi-objective test suite minimisation. *J. Syst. Softw.*, 83:689–701.

Yoo, S., Harman, M., and Ur, S. (2011a). Highly scalable multi objective test suite minimisation using graphics cards. In *Proceedings of the Third international conference on Search based software engineering*, SSBSE'11, pages 219–236, Berlin, Heidelberg. Springer-Verlag.

Yoo, S., Nilsson, R., and Harman, M. (2011b). Faster fault finding at google using multi objective regression test optimisation. In *8th European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 11), Szeged, Hungary*.

Young, M. and Pezze, M. (2005). *Software Testing and Analysis: Process, Principles and Techniques*. John Wiley & Sons.

Uma Abordagem de Análise do Tempo de Resposta para Teste de Desempenho em Aplicações Web

Priscila Guarienti, Maicon Bernardino, Avelino F. Zorzo e Flávio M. Oliveira

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

priscila.guarienti@acad.pucrs.br, bernardino@acm.org,

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

Abstract. *The testing activity is usually very expensive and time consuming, since most of the time it is executed manually. To improve that, Model-Based Testing (MBT) has been used lately to generate testing artifacts automatically. The main contribution of this work is to investigate how MBT can be applied for performance testing, i.e. how to include timing constraints in UML diagrams and based on them to execute the tests automatically, and verify whether the constraints are met or not. Another contribution is the setting of service level agreements, i.e. set parameters to verify whether a system delivers answers based on pre-agreed values. We applied our whole strategy to a complete case study and measured its execution time, as well as discuss some of our results.*

Resumo. *A atividade de teste é geralmente bastante onerosa, isso ocorre porque boa parte dessas atividades são executadas manualmente. Para melhorar isto, Model-Based Testing (MBT) tem sido utilizado para gerar artefatos de teste automaticamente. A principal contribuição deste trabalho é investigar como MBT pode ser aplicado em teste de desempenho, i.e. como incluir restrições de tempo em diagramas UML e com base nelas executar os testes e verificar se elas são satisfeitas ou não. Outra contribuição é a definição dos acordos de níveis de serviço, i.e. definir parâmetros para verificar se um sistema processa os tempos de respostas com base em valores pré-determinados. A abordagem proposta é aplicada a um estudo de caso real.*

1. Introdução

A evolução e o incremento da complexidade dos sistemas computacionais existentes têm tornado o processo de teste tão ou mais complexo que o processo de implementação do *software*. Para contornar este problema e aumentar a eficiência do processo de teste foram desenvolvidas diversas ferramentas com o objetivo de automatizar as atividades de verificação e validação (V&V).

Uma das abordagens que auxiliam o processo de teste de *software* é o teste baseado em modelos (*Model Based Testing - MBT*) [Apfelbaum and Doyle 1997], que tem o intuito de criar artefatos de teste que descrevem o comportamento do sistema. Neste modelo está incluso em sua especificação as características que serão testadas [El-Far and Whittaker 2001], com o objetivo de automatizar o processo de teste de

software. Esta abordagem consiste na geração dos cenários, casos e *scripts* de teste com base nos modelos do *software*. Além disso, o uso de MBT facilita as atividades desempenhadas pelos testadores, proporcionando maior eficácia e qualidade no processo de teste, reduzindo a probabilidade de má interpretação dos requisitos do sistema e/ou redução do tempo do teste.

Outro fator importante relacionado com a qualidade de *software* é o desempenho. Do ponto de vista do usuário final, um *software* possui bom desempenho quando apresenta um tempo de resposta aceitável, mesmo quando submetido a um volume de processamento próximo de situações reais [Meier et al. 2007]. Com o intuito de melhorar a qualidade do produto de *software*, a Engenharia de Desempenho de Software (*Software Performance Engineering - SPE*) [Woodside et al. 2007] aplica seus esforços para melhorar o desempenho de duas formas: no ciclo inicial baseado em modelos preditivos ou no ciclo final baseado em medições. Além disso, o teste de desempenho abrange atividades como: configurar o ambiente e o cenário de teste; identificar critérios de aceitação de desempenho; planejar e projetar os testes; executar os testes e analisar os resultados [Meier et al. 2007].

Neste contexto, a fim de aplicá-lo ao teste de desempenho, a abordagem proposta neste artigo enfatiza o MBT para avaliação de desempenho, como uma forma uniforme e formal de associar informações de tempo aos modelos UML. Posteriormente, com a execução dos casos de teste será possível coletar e analisar os resultados obtidos, além de comparar com os requisitos de tempo inicialmente estimados nos diagramas UML. Isso possibilita ter estimativas de tempo mais concretas e mensuráveis além de verificar a aceitabilidade do usuário em relação aos critérios de tempo estabelecidos nos acordos de níveis de serviço.

Além das contribuições nas áreas de MBT e teste de desempenho, a proposta deste trabalho contribui no desenvolvimento de uma Linha de Produto de *Software* (*Software Product Line - SPL*) [Clements and Northrop 2001] para geração de ferramentas de teste que usam MBT, chamada PLeTs (*Product Line for Model Based-testing tools*) [Rodrigues et al. 2010], [Silveira et al. 2011], [Costa et al. 2012]. De acordo com [Software Engineering Institute (SEI) 2014], SPL é importante e aplicável ao desenvolvimento de *software*, pois proporciona diversos benefícios durante o processo de desenvolvimento: menor custo, melhor qualidade, maior produtividade e maior velocidade de entrada em novos mercados. Isso é possível devido a flexibilidade que os componentes de *software* possuem na arquitetura da família possibilitando assim a reusabilidade.

Desta forma, este trabalho tem dois objetivos principais: auxiliar durante a fase de teste de *software*, prevendo situações e cenários que possam comprometer o desempenho de determinada aplicação e permitir a análise do tempo de resposta dos casos de teste. Para descrever estes aspectos foram definidos alguns estereótipos UML representando informações de tempo e que serão utilizados no processo de modelagem, geração e execução dos *scripts* de teste utilizando a PLeTs Perf [Silveira et al. 2011] que é uma ferramenta derivada da linha de produto PLeTs.

Este artigo está estruturado da seguinte forma. A Seção 2 faz uma breve introdução dos conceitos de teste de desempenho, SPL para MBT, técnica de avaliação e revisão de programas e acordo de nível de serviço. A Seção 3 apresenta detalhadamente a abor-

dagem deste trabalho. A Seção 4 apresenta um estudo de caso, bem como a análise e interpretação dos resultados. A Seção 5 discute alguns trabalhos relacionados. Ao final, a Seção 6 apresenta as considerações finais e trabalhos futuros.

2. Fundamentação Teórica

O teste de desempenho [Meier et al. 2007] tem por objetivo avaliar o comportamento do sistema submetido a uma determinada carga em um ambiente de teste específico. Ele fornece indicadores de desempenho que são utilizados para avaliar o quanto um sistema ou componente de um sistema é capaz de cumprir os requisitos de desempenho, tais como tempo de resposta ou *throughput*, além de identificar os possíveis gargalos que podem estar degradando o desempenho do sistema [Smith 2002].

Atualmente, está em desenvolvimento no Centro de Pesquisa em Engenharia de *Software* (CePES)[1] na PUCRS uma linha de produto de *software* denominada PLeTs [Rodrigues et al. 2010], que busca facilitar a derivação de ferramentas MBT, com as quais é possível executar casos de teste de forma automatizada. A relação da PLeTs com este trabalho diz respeito ao fato de que um dos produtos gerados por esta SPL refere-se a PLeTs Perf, o qual por meio da definição de transações e associação de tempo aos modelos UML serão definidos Acordos de Nível de Serviço (*Service Level Agreements - SLAs*) com o intuito de avaliar o desempenho das aplicações.

Um SLA caracteriza-se como um conjunto de procedimentos e objetivos formalmente acordados entre as entidades envolvidas com a finalidade de manter a qualidade de um serviço. Além de estipular métricas associadas para que o desempenho dos serviços e das aplicações possam ser medidos e avaliados [Blokdijk and Menken 2008].

Com base nas estimativas de tempo relacionadas as transações e por meio da Técnica de Avaliação e Revisão de Programas (*Program Evaluation and Review Technique - PERT*) [Stilian 1967] serão calculados os SLAs das transações. De acordo com Stilian a precisão das estimativas do tempo de execução de determinadas atividades podem ser aperfeiçoadas considerando as incertezas das estimativas e os riscos.

Neste contexto, esta técnica usa três estimativas de tempo para definir uma faixa aproximada para a execução de determinada atividade: I) Pessimista (T_P) - o tempo da atividade é com base na análise do pior cenário para a atividade; II) Mais Provável (T_M) - a duração da atividade, dados os prováveis recursos a serem designados, sua produtividade, expectativas realistas de disponibilidade para executar a atividade, dependências de outros participantes e interrupções; III) Otimista (T_O) - o tempo da atividade é com base na análise do melhor cenário para a atividade.

Com as estimativas de tempo estabelecidas a etapa seguinte consiste em calcular o tempo esperado (T_E) de execução de determinada atividade ou sequência de atividades. Este cálculo é obtido por meio de uma média ponderada por meio da seguinte fórmula:

$$T_E = \frac{(Pessimista + (4 * Mais\ Provavel) + Otimista)}{6}$$

Essa técnica é muito utilizada para planejar, avaliar e controlar o tempo de execução de programas e projetos. Após calcular o SLA, com base no cálculo do tempo

[1]Homepage CePES: http://www.cepes.pucrs.br

esperado, serão gerados os cenários e *scripts* de teste de desempenho que serão executados pela ferramenta LoadRunner [Hewlett Packard – HP 2014].

3. Abordagem Proposta

Em trabalhos anteriores [Silveira et al. 2011] [Costa et al. 2012] modelos e métodos para geração de sequências de teste de desempenho e teste estrutural foram investigados. Neste trabalho, será apresentada a aplicação de uma abordagem que tem por objetivo enfatizar o MBT, associando informações de tempo aos diagramas UML com o intuito de medir o tempo de execução das sequências do teste, bem como automatizar o processo de verificação dos requisitos de desempenho, *e.g.* tempo de resposta por meio dos SLAs.

A abordagem proposta de teste de desempenho é apresentada na Figura 1. Ela tem início com a divisão do processo de acordo com as ações que cada participante ou ferramenta envolvida realizará, *i.e.* `Tester`, `PLeTs` e `LoadRunner`. Este processo inicia com a atividade `Modelar os Diagramas UML`, desenvolvidos pelo `Tester` na especificação dos cenários de teste de desempenho, representado por diagramas de casos de uso (*Use Case Diagram - UC*) e por diagramas de atividades (*Activity Diagram - AD*). Com base no perfil UML SPT (*Schedulability, Performance and Time*) [Meier et al. 2007] são adicionadas aos modelos informações de teste relacionadas aos cenários de desempenho, cargas de trabalho (*workload*) e perfis dos usuários, representada pela atividade `Adicionar Dados de Teste de Desempenho`.

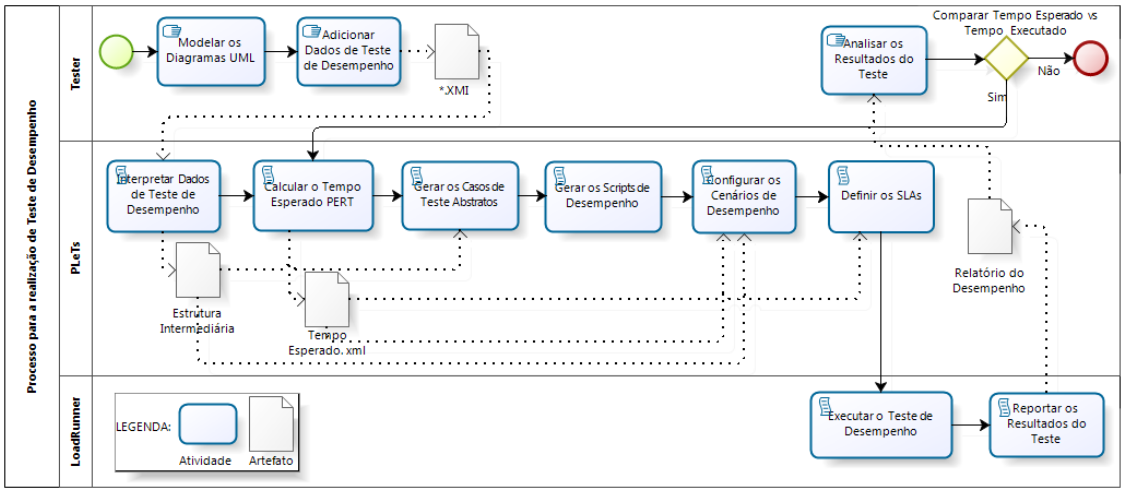


Figura 1. Processo da abordagem de teste de desempenho baseado em modelos

Visando a automação dos cenários e *scripts* de teste de desempenho pela `PLeTs`, o modelo UML gerado é exportado em um arquivo XMI (*XML Metadata Interchange*) que por sua vez é submetido ao *parser* da `PLeTs`, a fim de `Interpretar Dados de Teste de Desempenho` necessárias para a geração de uma estrutura de dados abstrata em memória (`Estrutura Intermediária`) que foram adicionadas por meio dos estereótipos.

Com o cenário de teste de desempenho especificado, cada caso de uso é decomposto em um AD, o qual demonstra o fluxo de ações executadas pelo usuário para realizar determinada tarefa no sistema. É nesta etapa que são definidas as transações que controlam as atividades e que serão necessárias para definir os SLAs e avaliar o tempo de

resposta dos *scripts* de teste de desempenho. É importante salientar que estimar tempo e esforço é ainda uma das atividades mais difíceis durante a fase de planejamento, isso inclui também as atividades de teste de *software*. No entanto, de alguma forma é necessário medir o esforço despendido para a realização dos casos de teste e avaliar se o resultado obtido está de acordo com o que foi estimado. Neste contexto e diante da dificuldade das equipes em estabelecer estimativas de tempo para a execução de testes de desempenho, os três pontos presentes na técnica PERT [Stilian 1967] (ver Seção 2), possibilitam uma faixa de variabilidade, sendo possível trabalhar com intervalos de tempo. A estimativa dos tempos diante das três perspectivas (pessimista, mais provável e otimista) estabelecidas pelo PERT é definida inicialmente a partir da execução de um *smoke test*[2]. Com base nesta técnica, a atividade seguinte é responsável por `Calcular o Tempo Esperado PERT` para cada uma das transações mapeados nos diagramas UML interpretados.

Em seguida, os diagramas UML com as informações de teste são convertidos em uma MEF (Máquina de Estados Finitos) e por meio da aplicação do método HSI (*Harmonized State Identification*) [Luo et al. 1995], são geradas as sequências de teste a serem executadas, representado pela atividade `Gerar os Casos de Teste Abstratos`.

As informações referentes a tecnologia que será usada são fornecidas nas etapas `Gerar Scripts de Desempenho` e `Configurar os Cenários de Desempenho`, quando o usuário deve fornecer todos os dados necessários para gerar os *scripts* e configurar os cenários de teste de desempenho usando uma ferramenta de teste específica, *i.e.* gerador de carga de trabalho. Na etapa a seguir é responsável por `Definir os SLAs` de acordo com as transações pré-definidas nos *scripts* de teste de desempenho gerados. Os cenários de teste de desempenho gerados serão usados posteriormente na atividade `Executar o Teste de Desempenho` pela ferramenta `LoadRunner` [Hewlett Packard – HP 2014].

Durante a execução dos *scripts* de teste de desempenho, o `LoadRunner` coleta os dados a fim de (`Reportar os Resultados do Teste`), que são gravadas em um arquivo XML denominado (`Relatório do Desempenho`) e reportados para a `PLeTs`. Desta forma é possível o `Tester` verificar as transações que cumpriram o SLA e as transações que não cumpriram, além de `Analisar os Resultados do Teste`, comparando o tempo mínimo, médio e máximo de resposta de cada transação, *i.e.* os tempos executados contra os tempos esperados.

As informações sobre cenários de teste de desempenho, carga de trabalho e perfil dos usuários são inseridas nos diagramas UC e AD por meio de estereótipos e rótulos (*tags*), que são necessários para geração dos *scripts* de teste de desempenho, bem como para avaliar a escalabilidade da aplicação em cenários com maiores requisições e concorrência. Neste contexto foram utilizados os estereótipos de desempenho conforme [de Oliveira et al. 2007] [Rodrigues et al. 2010] [Silveira et al. 2011] [Costa et al. 2012]: a) `TDtime`: determina o tempo total de execução do teste; b) `TDpopulation`: representa o número de usuários virtuais que irão executar a aplicação; c) `TDhost`: define o endereço (caminho) para executar a aplicação; d) `TDaction`: define o endereço do *link* acessado pelo ator; e) `TDmethod`: define o método de requisição HTTP utilizado pelo servidor; f) `TDparameters`: representa duas informações: nome e valor, que deve ser

[2]Processo no qual um produto de *software* é submetido, para verificar sua funcionalidade básica. Neste caso, será realizado um *smoke test* para coletar as estimativas de tempo que irão popular o modelo de teste.

preenchido para prosseguir, por exemplo, uma tela de login, ou um formulário de pesquisa; g) `TDprob`: indica a probabilidade de distribuição dos usuários virtuais em cada caso de uso. Esta informação deve estar presente em todas as associações *Actor-UseCase* e a soma de todos os rótulos `TDprob` anotados para um mesmo ator devem ser igual a 1 (ou seja, 100%). É importante ressaltar que o rótulo `TDprob` pode também ser informado nas transições existentes nos ADs com o objetivo de distribuir os usuários na execução de determinadas atividades da aplicação, como por exemplo em situações de paralelismo; h) `TDthinkTime`: denota o tempo existente entre a disponibilidade de execução da atividade até o início de sua real execução pelo usuário. Pode ser exemplificado pelo preenchimento de um formulário até sua submissão.

Estes estereótipos, bem como seus rótulos são apresentados nos exemplos nas Figuras 3 e 5. Como a maior contribuição deste estudo é gerar *scripts* de teste de desempenho considerando o tempo de resposta usando modelos UML, foi adicionado um novo estereótipo `TDexpTime`. Este estereótipo está presente nas transações do AD, e nele são informadas as estimativas de tempo referente à execução de cada transação. Caso esta informação não esteja presente no modelo, é executado um *smoke test* com o intuito de coletar os dados de teste de desempenho e gravar no arquivo XML `Tempo Esperado`.

Já a modelagem dos ADs é composta por duas etapas principais. A primeira etapa consiste na criação de um AD de alto nível, que na segunda etapa será expandido em outros ADs mais detalhados. Isto possibilita ter uma visão macro do processo de execução, além de permitir referenciar a um mesmo diagrama várias vezes, possibilitando, assim, o reuso dos modelos e dos *scripts* na execução dos testes de desempenho.

É importante ressaltar que o teste de desempenho também é composto por transações (*Transactions*) e requisições (*Requests*). Cada transação especifica um conjunto de requisições contidas em um *script* de teste de desempenho e são definidos com o intuito de isolar o processamento em diferentes pontos. Isto possibilita identificar os gargalos do sistema, além de mensurar as Transações por Segundo (TPS) e verificar se estas satisfazem os critérios de aceitação de acordo com as métricas estabelecidas no SLA.

O processo para definição dos SLAs inicia na criação dos modelos UML, em que são estabelecidas as transações e anotadas as estimativas de tempo para sua execução por meio do estereótipo `TDexpTime`. A partir do modelo UML criado um arquivo XMI é exportado. Em seguida, este arquivo XMI é interpretado e calculado o tempo de resposta esperado para execução de cada transação, por meio da técnica PERT. As informações do teste de desempenho, referente a criação do SLA, como nome da transação e estimativas do tempo ficam armazenadas em um arquivo XML, chamado `Tempo Esperado`, bem como o valor referente ao tempo de resposta esperado. Após calcular o valor esperado, tendo como referência os valores que estão presentes no arquivo XML, são gerados e executados os *scripts* de teste de desempenho, incluindo a criação dos SLAs no LoadRunner.

4. Estudo de Caso: Gerência de Portfólios e Projetos

Com o intuito de avaliar a abordagem proposta para teste de desempenho na perspectiva de inclusão de tempo nos modelos UML, esta seção detalha a realização de um estudo de caso, apresentando a atividade de modelagem, a geração dos cenários e *scripts* de teste de desempenho e sua instrumentalização para a ferramenta LoadRunner, juntamente com a definição dos SLAs, bem como análise dos resultados do teste. Para isso será

utilizada uma aplicação que tem por objetivo a gerência de portfólio de produtos e projetos multidisciplinares.

O cenário de testes de desempenho (Figura 2) é composto por quatro servidores da aplicação (*App Servers*), dois servidores *web* (*Web Servers*) e três servidores de banco de dados (*Database Servers*). Por meio da figura apresentada, também é possível verificar os aspectos de configuração de infraestrutura das máquinas, tais como: CPUs, memória e sistema operacional.

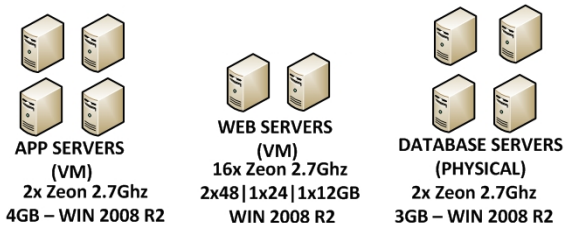


Figura 2. Ambiente do teste de desempenho

4.1. Modelos UML

O processo de modelagem tem início com a criação do cenário de teste de desempenho representado pelo UC (Figura 3). Nele é mostrada a interação do ator com os casos de uso com que ele está associado, tendo como base as principais funcionalidades da aplicação.

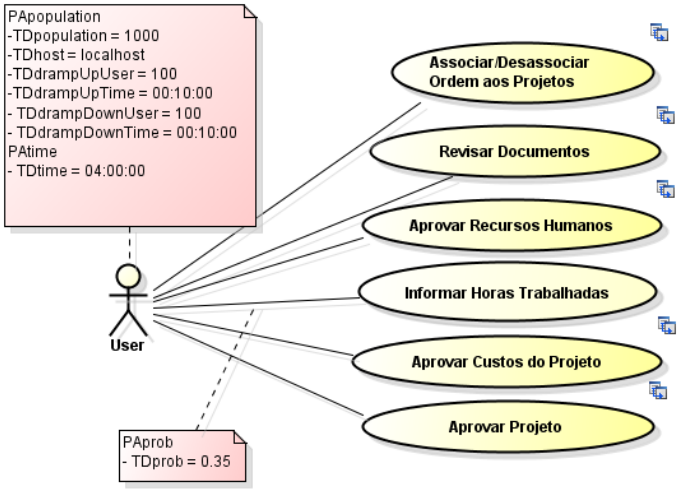


Figura 3. Diagrama de casos de uso da aplicação

Para a execução dos testes foi modelado um cenário de carga de trabalho composto por 1000 usuários virtuais, sendo que durante a inicialização do teste são acrescentados 100 usuários (`TDrampUpUser`) a cada 10 minutos (`TDrampUpTime`), totalizando 1h40min para a inicialização total do teste. A finalização do teste (`TDrampDownUser` e `TDrampDownTime`) também foi configurada com os mesmos parâmetros. Totalizando 3h20min entre a inicialização e finalização do teste, resultando em 40min de execução do teste com os 1000 usuários virtuais concorrentes para uma duração total de execução do teste de 4h (`TDtime`).

Após definido o cenário de teste, a próxima etapa de elaboração do modelo de teste é decompor cada caso de uso em um AD. Sendo assim, foram construídos ADs correspondentes aos casos de uso apresentados na Figura 3. Outra característica importante é o parâmetro `TDprob` que define a probabilidade de distribuição dos usuários virtuais em cada caso de uso relacionado a ele. Para representar este processo será exemplificado o caso de uso `Informar Horas Trabalhadas`[3]. A probabilidade do ator em executar este caso de uso é equivalente a `TDprob` = 35%, o qual representa 350 usuários virtuais dos 1000 concorrentes, conforme também é apresentado na Tabela 1, assim como os demais casos de uso que somados totalizam a carga de trabalho dos usuários virtuais no cenário de desempenho avaliado.

Tabela 1. Distribuição da carga no cenário de teste de desempenho da aplicação

Ator	Caso de Uso	Probabilidade	Usuários Virtuais
User	Associar/Desassociar Ordem aos Projetos	20%	200
	Informar Horas Trabalhadas	35%	350
	Revisar Documentos	20%	200
	Aprovar Recursos Humanos	10%	100
	Aprovar Custos do Projeto	10%	100
	Aprovar Projeto	5%	50
			1000

Nesta etapa inicialmente é criado um AD alto nível, identificando todas as atividades que serão necessárias para a criação e execução do *script* de teste de desempenho. Conforme observa-se na Figura 4, o AD é composto por três atividades principais, sendo que a primeira e a terceira são referentes as atividades de `login` e `logout`, e a do meio é referente a atividade `Informar Horas Trabalhadas`. É importante salientar que neste diagrama não existem estereótipos e informações nas atividades e nas transições, ele serve apenas para demonstrar a sequência de execução dos demais diagramas que compõem o cenário de teste e auxiliar no gerenciamento e reuso dos modelos.

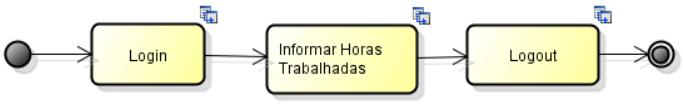


Figura 4. Diagrama de atividades do caso de uso `Informar Horas Trabalhadas`

Em seguida, cada uma das três atividades é expandida em um novo diagrama, que possuirá as informações de teste, anotadas por meio dos estereótipos apresentados na Seção 3. Já as estimativas de tempo diante das três perspectivas (pessimista, mais provável e otimista) são anotadas nas transações existentes no diagrama, conforme é possível observar na Figura 5.

O diagrama mostrado na Figura 5 representa a interação do usuário no sistema no momento em que o mesmo irá informar o tempo consumido em cada tarefa relacionada ao projeto. Esta ação é composta por três transações: na transação `Acessar Folha Ponto` estão presentes as ações de acesso à "Página Inicial" do sistema, selecionando em seguida as opções "Aplicação" e "Folha Ponto"; na transação `Folha Ponto Projeto` são

[3]Todos os diagramas UML modelados para o estudo de caso Gerência de Portfólios e Projetos estão disponíveis no endereço http: //www.cepes.pucrs.br/performance_time

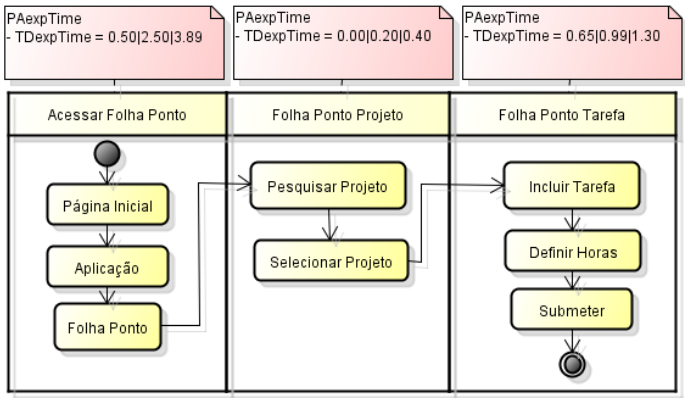


Figura 5. Subdiagrama de atividades da atividade `Informar Horas Trabalhadas`

executadas as ações "Pesquisar Projeto" e "Selecionar Projeto"; finalizando, na transação `Folha Ponto Tarefa` é composta pelas atividades "Incluir Tarefa", "Definir Horas" e "Submeter" os dados. Por meio do estereótipo `TDexptime`, anotado em cada transação, é possível verificar os tempos estimados para execução de cada sequência de teste. Por exemplo, na transação `Acessar Folha Ponto` foram estimados os seguintes tempos de execução: mínimo: 0.50, médio: 2.50 e máximo: 3.89. É importante salientar que este processo foi definido para os ADs da atividade `Login` e `Logout`, presentes na Figura 4, como também para os demais casos de uso representados pela Figura 3. Por meio das transações definidas nos ADs apresentados e das estimativas de tempo de resposta informadas é que serão definidos os SLAs para cada transação. Posteriormente, estas transações serão interpretadas pela ferramenta LoadRunner na execução dos cenários e *scripts* de teste de desempenho, conforme descrito na Seção 3.

4.2. Geração dos cenários e *scripts* de teste de desempenho

Os diagramas UML modelados na Seção 4.1 são representados por um arquivo XMI com as informações de teste. Essas informações por sua vez são submetidas a PLeTs a fim de convertê-las em um modelo formal, *e.g.*, Máquinas de Estados Finitos (MEFs).

Primeiramente, o modelo é analisado e convertido em uma estrutura de dados correspondente aos diagramas UML. A próxima etapa consiste em converter este modelo UML em uma nova estrutura genérica, chamada de estrutura intermediária que fica armazenada em memória, a qual será usada como entrada pelos diferentes modelos formais, *e.g.*, MEFs. Posteriormente, os modelos UML são convertidos em MEFs, e por meio da aplicação do método de geração de sequências de teste HSI [Luo et al. 1995] [Costa et al. 2012], são gerados os casos de teste abstratos. Finalmente, estes casos de teste abstratos são instanciados para a ferramenta LoadRunner a fim de que sejam gerados os cenários e *scripts* de teste de desempenho.

A Figura 6, apresenta um trecho do código XML, referente ao *script* de teste de desempenho gerado pela PLeTsPerf. Conforme pode-se observar, ele é composto pelas diversas requisições (*requests*), referente as ações realizadas pelo usuário no momento de incluir uma tarefa no projeto. Com base nos tempos estimados nos modelos é calculado o tempo esperado em segundos (*Threshold* = 0.985), que definirá o SLA dentro do arquivo de configuração do cenário de teste de desempenho. Por limitações de espaço foi apresentado somente um trecho do *script* LoadRunner, no entanto, nele estão presentes

todas as transações e atividades definidas no diagrama. Vale destacar, que com base nas transações presente no *script* e no tempo de resposta esperado configurado no cenário, neste momento também são criadas os SLAs.

```
1          lr_start_transaction("Folha Ponto Tarefa")
2            lr_start_sub_transaction("Incluir Tarefas")
3              web_url("Incluir Tarefas",
4              "URL=http://{servidor}/folha_ponto/projeto/tarefa/incluir_tarefa",
5              "Method=GET",
6              "Resource=0",
7              "Referer=http://{servidor}/folha_ponto/projeto/tarefa/incluir_tarefa",
8              "Mode=HTTP",
9              ITEMDATA,
10               "Name=PROJETO", "Value={{LISTA_PROJETOS}}", ENDITEM,
11               "Name=TAREFA", "Value={{LISTA_TAREFAS}}", ENDITEM,
12               "Name=UNIDADE_NEGOCIO", "Value={{LISTA_NEGOCIOS}}", ENDITEM,
13             LAST);
14           lr_end_sub_transaction("Incluir Tarefas",LR_AUTO)
15           lr_think_time(5);
16         lr_end_transaction("Folha Ponto Tarefa",LR_AUTO);
```

Figura 6. Trecho de Script LoadRunner da transação `Folha Ponto Tarefa`

Ao término da execução, o LoadRunner apresenta um resumo, demonstrando os tempos de resposta de cada transação, além do *status* de quais as transações que passaram no teste e quais as transações que falharam de acordo com o SLA definido, possibilitando assim, comparar e analisar os resultados obtidos com a realização do teste. É importante ressaltar que as informações referente a execução dos testes, bem como as transações e estimativas de tempo também são armazenadas em um arquivo XML chamado de `Expected Time`, conforme descrito na Seção 3 e são reportadas para a PLeTsPerf. Sendo assim, o usuário poderá além de analisar os resultados, interagir com a aplicação calibrando o modelos de acordo com o cenário e a carga que deseja testar, tendo como base os próprios resultados de execução dos testes.

4.3. Análise e Interpretação dos Resultados

Conforme é possível observar no gráfico apresentado na Figura 7 em que destaca a evolução do tempo de resposta em relação a cada uma das cargas de trabalho submetidas, essa transação compreende o conjunto de atividades apresentados na Figura 5 que faz referência ao caso de uso `Informar Horas Trabalhadas`. A figura apresenta os tempos de resposta para cada carga de trabalho, os tempos estimados de acordo com a técnica PERT (Pessimista, Mais Provável e Otimista), bem como o SLA definido pelo tempo esperado, e ainda os tempos executados do teste de desempenho, destacando o tempo Mínimo, Médio e Máximo coletados pelo LoadRunner.

Analisando o gráfico da Figura 7 pode-se evidenciar que a carga de 100 usuários obteve os tempos médios e mínimos muito inferiores às demais cargas, inclusive com um desvio superior ao SLA estimado. Entretanto, as demais cargas obtiveram tempos médios próximos aos SLAs definidos. Vale destacar que a carga de 300 usuários virtuais em que o tempo do SLA estimado foi de 1.67s contra o tempo médio executado pelo teste de desempenho de 1.60s, essa foi a menor diferença entre as estimativas de tempo (SLA) e o tempo executado, variando apenas 4.2%.

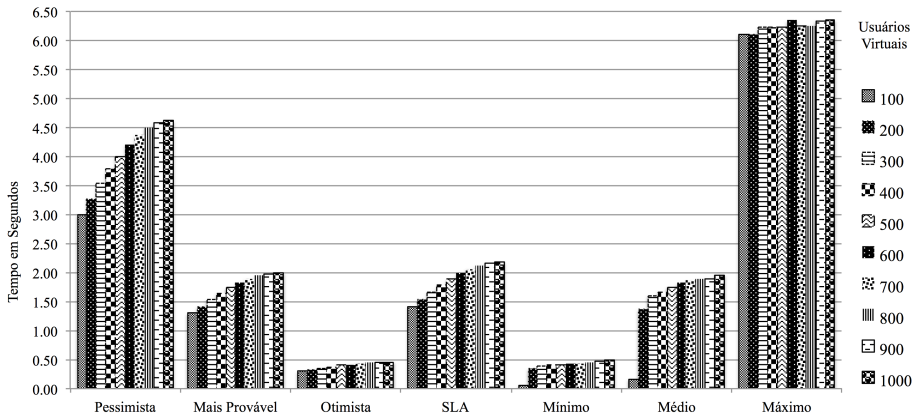


Figura 7. Gráfico de análise do tempo de resposta da transação Informar Horas Trabalhadas do diagrama de atividades da Figura 4

Com a realização do teste de desempenho, pode-se observar também que dependendo do aumento dos usuários virtuais que acessam o sistema, o SLA deverá ser revisto. Pois, à medida que mais usuários acessavam a aplicação, o tempo de resposta das sequências de teste tendem a aumentar e as transações que inicialmente cumpriam com o valor estabelecido no SLA passam a não mais cumprir, exceto nos casos em que a solução algorítmica seja linear. Outro fator que também deve ser considerado quando o SLA é definido é o ambiente em que a aplicação será executada, além dos históricos de execuções que permitam verificar o tempo de resposta das transações, permitindo assim, definir estimativas mais concretas e próximas ao ambiente real de execução dos testes de desempenho. Isso pode ser observado na Figura 7 apresentado, uma vez que a medida que novas execuções eram realizadas o valor estimado para o SLA ficava mais próximo do tempo médio, o que não acontece com a carga de 100 usuários, em que os tempos são estimados sem considerar execuções anteriores.

5. Trabalhos Relacionados

Atualmente, há um interesse crescente relacionado à área de teste de desempenho. Neste contexto, diversos autores têm apresentado abordagens e metodologias para testar o desempenho das aplicações utilizando técnicas de teste aplicadas no desenvolvimento de *software*. Em particular, teste de desempenho com modelos UML é um dos tópicos mais abordados pelos pesquisadores. Diversos trabalhos apresentam abordagens e metodologias que derivam cenários e *scripts* de teste de desempenho a partir de modelos UML adaptados para representar informações de desempenho, *e.g.*, UCs, ADs, etc. Alguns dos principais trabalhos nesta área são apresentados em [Kalita et al. 2011], [Geetha et al. 2011], [Vasar et al. 2012] e [Weiss et al. 2013].

Na abordagem proposta em [Kalita et al. 2011], é apresentada uma ferramenta chamada PReWebN (*Prototype Research Web application in Netbeans platform*) com o objetivo de avaliar o desempenho de aplicações com base em tecnologia Java. A ferramenta realiza uma análise estatística a partir dos dados coletados pela ferramenta de geração de carga de trabalho para diferentes métricas de desempenho, entre elas o tempo de resposta (*response time*). No entanto, a abordagem proposta pela autora não se baseia na abordagem MBT. Portanto, tal abordagem não se beneficia das vantagens da adoção

de MBT, como a validação de requisitos de desempenho por meio de modelos de teste, geração sistemática e automação dos SLAs.

Em [Geetha et al. 2011] um *framework* híbrido para um modelo de processo de predição de desempenho é apresentado, a fim de avaliar o desempenho de aplicações já na fase de estudo de viabilidade do projeto de *software*. Os autores propõem uma abordagem de engenharia de desempenho com base em diagramas de casos de uso UML, oferecendo flexibilidade para integrar o processo de previsão de desempenho com processos de engenharia de *software*. Apesar do estudo ser pautado na abordagem MBT, o *framework* proposto não leva em consideração a geração automática e análise dos SLAs.

Um *framework* usando uma heurística sensível ao tempo de resposta é apresentado em [Vasar et al. 2012] . Este *framework* serve como base para suportar a alocação dinâmica de servidores em ambiente de computação em nuvem, dada a carga de processamento exigido pela aplicação. Apesar do estudo levar em consideração a automação dos SLAs, os mesmos não são previamente definidos com base em modelos (MBT) e integrados a ferramentas de desempenho, tal como a abordagem proposta neste artigo, mas sim avaliados por meio da integração de ferramentas de gerenciamento da escalabilidade em ambientes de computação em nuvem. Por fim, em [Weiss et al. 2013] uma abordagem para avaliação sistemática de desempenho com base em aplicações de referência (*benchmark*) adaptadas é descrita. Assim como os demais trabalhos apresentados nesta seção, os autores também não contemplam em sua abordagem a automação dos SLAs como proposto pelo presente artigo.

Embora estes trabalhos apresentem alternativas que contribuem para minimizar o esforço na atividade de teste de desempenho, a maioria foca somente em avaliar as métricas de desempenho. Em geral, a maioria destas pesquisas deixa muitas lacunas quando se referem a automação dos SLAs de desempenho. Por outro lado, a abordagem proposta neste artigo também busca minimizar este esforço por meio da abordagem MBT para geração e avaliação sistemática do tempo de resposta usando SLAs.

6. Considerações Finais

Este trabalho apresentou uma abordagem de "Análise do tempo de resposta para teste de desempenho em aplicações *Web*", tendo como principal objetivo medir o tempo de execução das sequências de teste. Inicialmente, o processo de modelagem apresenta as informações necessárias para a realização de teste de desempenho que servem como entrada para as ferramentas de teste baseada em modelos, que são geradas a partir da linha de produto - PLeTs. Com base neste modelo as ferramentas MBT geram casos e *scripts* de teste de desempenho para um gerador de carga, *e.g.* LoadRunner. Para demonstração da abordagem proposta no trabalho foi desenvolvido um estudo de caso, em que foram criados cenários e *scripts* de teste de desempenho para a ferramenta LoadRunner, medindo o tempo de execução de cada transação em relação aos tempos definidos no SLA.

Após criar o SLA, o teste foi executado, e com os resultados obtidos com a execução foi realizada a etapa de análise dos resultados. Nesta etapa, foi possível identificar, de acordo com o cenário criado e os tempos definidos nos SLAs quais as transações que passaram e quais falharam no teste, além de comparar os tempos estimados em relação ao tempo executado por cada transação. Outro aspecto importante a ser considerado, é que uma vez criados os modelos, os tempos estimados que são informados, podem ser

ajustados de acordo com o cenário de desempenho que se deseja testar, tendo como base os históricos obtidos com as execuções anteriores. Também é importante ressaltar que os *scripts* de teste de desempenho gerados foram executados pelo LoadRunner. No entanto, a abordagem proposta foi desenvolvida com o objetivo de ser estendida para outras ferramentas de desempenho, como por exemplo, o VisualStudio. Da mesma forma que os produtos gerados pela PLeTs não compreendem um módulo de análise dos resultados que até então foram analisados manualmente, sendo essa uma das atividades de trabalhos futuros, permitindo uma integração maior com as tecnologias geradoras de carga de trabalho.

Diante da abordagem proposta, e na realização de um estudo de caso baseado em uma aplicação de gerenciamento de portfólios e projetos, foi possível demonstrar todas as etapas realizadas para a execução de teste de desempenho. Descrevendo desde a fase de modelagem UML até a definição dos SLAs e instrumentalização dos *scripts* de teste de desempenho gerados para a ferramenta LoadRunner. Além disso, outro aspecto significativo na realização do trabalho é que os dados obtidos com a execução dos testes são referentes a uma aplicação amplamente utilizada pela indústria, o que proporciona um cenário e um ambiente de teste muito similar a situações reais.

7. Agradecimentos

Avelino Zorzo, Flávio Oliveira, Maicon Bernardino e Priscila Guarienti são pesquisadores do Centro de Competência em Teste de Desempenho da PUCRS, uma parceria entre a Dell Computadores do Brasil Ltda e PUCRS. Estudo desenvolvido pelo Grupo de Pesquisa do PDTI 001/2014, financiado pela Dell Computadores do Brasil Ltda com recursos da Lei 8.248/91. Agradecemos também aos demais membros do grupo de pesquisa do CEPES, Elder de Macedo Rodrigues e Leandro Teodoro Costa pelas discussões sobre a PLeTs.

Referências

Apfelbaum, L. and Doyle, J. (1997). Model Based Testing. In *Proceedings of the Software Quality Week Conference*, pages 296–300.

Blokdijk, G. and Menken, I. (2008). *Service Level Management Best Practice Handbook: Building, Running and Managing Effective Service Level Management SLAs - Ready to Use Supporting Documents Bringing ITIL Theory into Practice*. Emereo Pty Ltd, London, UK.

Clements, P. and Northrop, L. (2001). *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing, 3rd edition.

Costa, L. T., Czekster, R. M., de Oliveira, F. M., Rodrigues, E. M., Silveira, M. B., and Zorzo, A. F. (2012). Generating Performance Test Scripts and Scenarios Based on Abstract Intermediate Models. In *Proceedings of the 24th International Conference on Software Engineering and Knowledge Engineering*, pages 112–117.

de Oliveira, F. M., Menna, R. d. S., Vieira, H. V., and Ruiz, D. D. A. (2007). Performance Testing from UML Models with Resource Descriptions. *I Brazilian Workshop on Systematic and Automated Software Testing*, pages 47–51.

El-Far, I. K. and Whittaker, J. A. (2001). *Model-based Software Testing*, pages 825–837. Wiley, New York, USA.

Geetha, D. E., Kumar, T. S., and Kanth, K. R. (2011). Framework for Hybrid Performance Prediction Process Model: Use Case Performance Engineering Approach. *SIGSOFT Software Engineering Notes*, 36(3):1–15.

Hewlett Packard – HP (2014). Software HP LoadRunner. Disponível em: https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp.

Kalita, M., Khanikar, S., and Bezboruah, T. (2011). Investigation on performance testing and evaluation of PReWebN: a java technique for implementing web application. *Institution of Engineering and Technology Software*, 5(5):434–444.

Luo, G., Petrenko, A., and v. Bochmann, G. (1995). Selecting test sequences for partially-specified nondeterministic finite state machines. In *7th IFIP WG 6.1 International Workshop on Protocol Test Systems*, pages 95–110.

Meier, J., Farre, C., Bansode, P., Barber, S., and Rea, D. (2007). *Performance testing guidance for web applications: patterns & practices*. Microsoft Press, Redmond, USA.

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

Silveira, M. B., Rodrigues, E. M., Zorzo, A. F., Costa, L. T., Vieira, H. V., and de Oliveira, F. M. (2011). Generation of Scripts for Performance Testing Based on UML Models. In *Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering*, pages 258–263.

Smith, C. U. (2002). *Software Performance Engineering*. John Wiley & Sons, New York, USA.

Software Engineering Institute (SEI) (2014). Software Product Lines (SPL). Disponível em: http://www.sei.cmu.edu/productlines/.

Stilian, G. (1967). *PERT: un nuevo instrumento de planificación y control*. Deusto.

Vasar, M., Srirama, S. N., and Dumas, M. (2012). Framework for Monitoring and Testing Web Application Scalability on the Cloud. In *Proceedings of the WICSA/ECSA Companion Volume*, pages 53–60.

Weiss, C., Westermann, D., Heger, C., and Moser, M. (2013). Systematic Performance Evaluation Based on Tailored Benchmark Applications. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, pages 411–420.

Woodside, M., Franks, G., and Petriu, D. C. (2007). The Future of Software Performance Engineering. In *Proceedings of the Future of Software Engineering*, pages 171–187.

Uma Estratégia de Testes Logarítmica para o Algoritmo Hi-ADSD

Vinicius K. Ruoso, Luis Carlos E. Bona, Elias P. Duarte Jr.

[1]Departamento de Informática – Universidade Federal do Paraná (UFPR)
Caixa Postal 19.081 – 81.531-980 – Curitiba – PR – Brasil

{vkruoso,bona,elias}@inf.ufpr.br

Abstract. *The* Hierarchical Adaptive Distributed System-level Diagnosis (Hi-ADSD) *is a distributed diagnosis algorithm that creates a virtual topology based on a hypercube. A hypercube is a scalable structure by definition, presenting important topological features like: symmetry, logarithmic diameter and good fault tolerance properties. The latency of the algorithm executed on a system with* N *nodes is at most* $\log_2^2 N$ *However, the number of executed tests in the worst case is quadratic. This work presents a new testing strategy for the* Hi-ADSD *algorithm, which guarantees that at most* $N \log_2 N$ *tests are executed at each* $\log_2 N$ *testing rounds. The maximum latency is mantained in* $\log_2^2 N$ *testing rounds. The algorithm is adapted to the new testing strategy. Furthermore, the use of* timestamps *is adopted to allow each node to retrieve diagnosis information from several other nodes, thus reducing the average latency. The new algorithm is specified, formal proofs are given, and experimental results obtained by simulations are presented and compared with the* Hi-ADSD *algorithm.*

Resumo. *O algoritmo* Hierarchical Adaptive Distributed System-level Diagnosis (Hi-ADSD) *é um algoritmo de diagnóstico distribuído que cria uma topologia virtual baseada em um hipercubo. O hipercubo é uma estrutura escalável por definição, apresentando características topológicas importantes como: simetria, diâmetro logarítmico e boas propriedades para tolerância a falhas. A latência do algoritmo quando executado em um sistema com* N *nodos é de no máximo* $\log_2^2 N$ *rodadas de teste. Entretanto, o número de testes executados no pior caso é quadrático. Este trabalho apresenta uma nova estratégia de testes para o algoritmo* Hi-ADSD *que garante que são executados no máximo* $N log_2 N$ *testes a cada* $log_2 N$ *rodadas. A latência máxima é mantida em* $\log_2^2 N$ *rodadas de teste. O algoritmo é adaptado para a nova estratégia de testes. Além disso, foi adotado o uso de* timestamps *para permitir que cada nodo obtenha informações de diagnóstico a partir de diversos outros nodos, consequentemente reduzindo a latência média. O novo algoritmo é especificado, suas provas formais são demonstradas e resultados experimentais obtidos por simulações são apresentados e comparados com o* Hi-ADSD.

1. Introdução

Considere um sistema composto de N nodos, cada um destes nodos pode assumir um de dois estados: falho ou sem-falha. Um algoritmo de diagnóstico distribuído em nível de sistema [Duarte Jr. et al. 2011, Masson et al. 1996] permite que os nodos sem-falha do

sistema determinem o estado de todos os outros nodos. Nodos desse sistema devem ser capazes de executar testes em outros nodos, podendo assim determinar em qual estado o nodo testado está. Assume-se que um testador sem-falha consegue determinar corretamente o estado do nodo testado.

Esta asserção referente à capacidade de um nodo sem-falha realizar testes com precisão faz parte do primeiro modelo proposto na área de diagnóstico em nível de sistema, apresentado por Preparata, Metze e Chien [Preparata et al. 1967], chamado modelo PMC. Segundo o modelo, o diagnóstico clássico consiste na escolha de um conjunto de testes a serem realizados e na obtenção dos resultados destes testes para o diagnóstico do sistema.

O diagnóstico adaptativo [Hakimi and Nakajima 1984], por sua vez, sugere que uma unidade determine quais testes vai executar baseando-se nos resultados dos testes executados previamente. Desta forma, cada unidade adapta o conjunto de testes que irá realizar de acordo com o resultado de testes anteriores. No diagnóstico adaptativo uma rodada de testes é definida como o tempo necessário para que as unidades sem-falha do sistema executem seus testes assinalados.

O modelo PMC considera a existência de um observador central, que é livre de falhas, para a realização do diagnóstico do sistema. Unidades realizam testes e enviam seus resultados a este observador, que os analisa e conclui qual o estado de todas as unidades do sistema. O diagnóstico distribuído em nível de sistema [Hosseini et al. 1984] é uma abordagem onde este observador central não existe, e cada unidade é responsável por realizar o diagnóstico do sistema como um todo.

Para que seja possível comparar o desempenho de diferentes algoritmos de diagnóstico são utilizadas duas métricas: latência e quantidade de testes. Em um sistema diagnosticável a ocorrência de um evento em uma unidade leva um certo tempo para ser detectada pelas outras unidades. A latência é definida como o número de rodadas de testes necessárias para que todos os nodos do sistema realizem o diagnóstico de um evento ocorrido. Define-se um evento como sendo a troca de estado de uma unidade do sistema, tanto de falho para sem-falha quanto de sem-falha para falho. Outra métrica importante para a avaliação de desempenho de estratégias de diagnóstico é a quantidade máxima de testes que o sistema realiza em uma rodada de testes.

Em [Duarte Jr. and Nanya 1998] o algoritmo *Hierarchical Adaptive Distributed System-Level Diagnosis (Hi-ADSD)* é introduzido. Este algoritmo apresenta uma latência de diagnóstico de no máximo $\log_2^2 N$ rodadas de teste para um sistema de N nodos. Os nodos são organizados em *clusters* progressivamente maiores. Os testes são executados de uma maneira hierárquica, inicialmente no *cluster* com dois nodos, indo para o *cluster* com quatro nodos, e assim sucessivamente, até que o *cluster* com $N/2$ nodos seja testado. Para que o nodo receba informações sobre um dado *cluster*, o nodo executa testes até que ele encontre um nodo não falho ou teste todos os nodos do *cluster* falhos. Considerando a rede como um todo, o maior número de testes executados em uma rodada de testes é $N^2/4$. Este cenário acontece quando $N/2$ nodos de um mesmo *cluster* estão falhos, e os outros $N/2$ nodos testam este *cluster* na mesma rodada de testes.

Neste trabalho apresentamos uma nova estratégia de testes para o algoritmo *Hi-ADSD*. Esta estratégia requer menos testes por rodadas de teste no pior caso, e mantém a

latência de $\log_2^2 N$ rodadas. Na estratégia original utilizada pelo *Hi-ADSD* para determinar seus testes, um nodo pode ser testado por vários outros nodos. A ideia principal da nova estratégia de testes é evitar que estes testes extras sejam executados. Para isso, uma ordem de testes é estabelecida para cada nodo em cada *cluster*. Assim, para um nodo i fazer um teste no nodo j na rodada de testes s, o nodo i deve ser o primeiro nodo sem-falha na lista de nodos testadores de j na rodada s. Desta maneira conseguimos evitar múltiplos testes em um mesmo nodo. Como cada nodo recebe no máximo um teste por rodada de testes, a nova estratégia garante um número de testes executados no sistema de no máximo $N \log_2 N$ a cada $\log_2 N$ rodadas de teste, fato que é provado no trabalho. Resultados de simulação são apresentados comparando o algoritmo *Hi-ADSD* original com a nova estratégia, tanto em termos de latência como do número de testes executados.

O restante deste trabalho está organizado da seguinte forma. A seção 2 apresenta conceitos e trabalhos relacionados. A seção 3 descreve a nova estratégia de testes. Por fim, a seção 4 conclui o trabalho.

2. Diagnóstico Distribuído Hierárquico

Considere um sistema S que consiste em um conjunto de N nodos, $n_0, n_1, ..., n_{N-1}$, alternativamente o nodo n_i pode ser chamado de nodo i. Assumimos que o sistema é totalmente conectado, ou seja, existe um canal de comunicação entre qualquer par de nodos e estes canais nunca falham. Cada nodo n_i pode estar falho ou sem-falha. Os nodos do sistema são capazes de testar outros nodos e determinar seu estado corretamente. O objetivo do diagnóstico é determinar o estado de todos os nodos do sistema. Neste trabalho assumimos que falhas são apenas do tipo *crash*, i.e. nodos falhos não interagem com outros nodos do sistema. Este capítulo apresenta conceitos e algoritmos de diagnóstico, focando em particular nos algoritmos hierárquicos.

2.1. O Algoritmo *Hi-ADSD*

O algoritmo *Hierarchical Adaptive Distributed System-Level Diagnosis (Hi-ADSD)* [Duarte Jr. and Nanya 1998] é hierárquico, além de ser distribuído e adaptativo, e realiza testes baseados em uma estratégia de dividir para conquistar. Esta hierarquia é baseada em um hipercubo. Quando todos os nodos do sistema estão sem-falha o grafo de testes é um hipercubo completo.

O hipercubo [LaForge et al. 2003] é uma estrutura escalável por definição, apresentando características topológicas importantes como: simetria, diâmetro logarítmico e boas propriedades para tolerância a falhas. Em um hipercubo, um nodo i é vizinho de um nodo j se, e somente se, os identificadores de i e j têm apenas um *bit* diferente em suas representações binárias. Dizemos que um hipercubo com 2^d elementos tem dimensão d. Em um hipercubo de dimensão d, um caminho entre quaisquer dois nodos tem no máximo d arestas. A Figura 1 mostra um 3-hipercubo.

No *Hi-ADSD* os nodos são organizados em *clusters* para execução de testes e obtenção de informações de diagnóstico. O tamanho dos *clusters* é sempre uma potência de dois, e baseia-se na estrutura do hipercubo. No primeiro intervalo de testes um nodo testa outro nodo e vice-versa, formando um *cluster* de dois nodos. No segundo intervalo de testes o testador testa um *"cluster"* de dois nodos, isto é, testa um nodo e obtém informação sobre o outro. No terceiro intervalo de testes o testador testa um cluster de

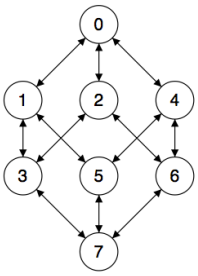


Figura 1. Um 3-hipercubo.

quatro nodos, e assim sucessivamente até que no *d*-ésimo intervalo de testes o testador testa um *cluster* com $N/2$ nodos, e todos formam um *cluster* de N nodos, ou seja, o sistema todo. A Figura 2 mostra os *clusters* formados em um sistema com 8 nodos.

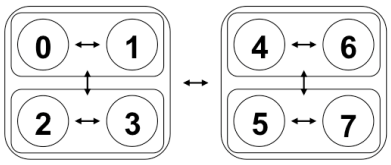


Figura 2. Clusters formados pelo algoritmo *Hi-ADSD* em um sistema de 8 nodos.

A lista ordenada de nodos testados por um determinado nodo i em um *cluster* de tamanho 2^{s-1} é dada pela função $c_{i,s}$. A Equação 1 mostra a especificação do algoritmo para esta função. Definimos o símbolo $\oplus$ como a operação *XOR*. Um exemplo de uso desta função em um sistema com 8 nodos é exposto na Tabela 1.

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

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	4,6,7,5	5,7,6,4	6,4,5,7	7,5,4,6	0,2,3,1	1,3,2,0	2,0,1,3	3,1,0,2

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

Quando um testador testa um nodo sem-falha, ele obtém informações de diagnóstico sobre os demais nodos do *cluster* do nodo testado. Quando o testador testa um nodo falho, ele continua testando nodos daquele *cluster* até encontrar um nodo não falho ou testar todos os nodos do *cluster* como falhos. Por exemplo, em um sistema com 8 nodos inicialmente o nodo 0 testa o nodo 1 do *cluster* de tamanho 1, na próxima rodada de testes testa o *cluster* de tamanho 2 composto pelos nodos 2 e 3, por fim testa o *cluster* de tamanho 4 que contém os nodos 4, 5, 6 e 7 e então retorna a testar o *cluster* de tamanho 1, e assim sucessivamente.

O Algoritmo 1 mostra em pseudo-código o funcionamento do algoritmo *Hi-ADSD* executado no nodo i. O algoritmo realiza testes sequencialmente nos nodos retornados pela função $c_{i,s}$ até testar um nodo sem-falha. Ao testar um nodo sem-falha obtém-se

informações de diagnóstico sobre os outros nodos daquele *cluster*. A latência do algoritmo *Hi-ADSD* é de no máximo $\log_2^2 N$ rodadas de teste.

Esta estratégia de testes pode gerar um número quadrático de testes. Por exemplo, suponha que um sistema de N nodos está com $N/2$ nodos falhos, e que tais nodos formam um *cluster* com $N/2$ nodos. Portanto, este é um sistema onde existe um *cluster* de $N/2$ nodos falhos e outro de $N/2$ nodos não falhos. Seguindo a estratégia acima, suponha que todos os nodos do *cluster* não falho estão na rodada de testes em que irão testar outro *cluster* de tamanho $N/2$. Assim, cada nodo não falho irá realizar $N/2$ testes, resultando em um total de $N/2 * N/2 = N^2/4$, ou seja, O(N^2) testes em uma rodada de testes. Esta mesma complexidade pode ser obtida com um algoritmo de força bruta.

```
1  Algoritmo Hi-ADSD executado no nodo i
2  repita
3      para s ← 1 até log_2 N faça
4          repita
5              node_to_test := próximo nodo em c_{i,s}
6              teste(node_to_test)
7              se node_to_test está sem-falha então
8                  atualiza informação de diagnóstico do cluster
9          até node_to_test está sem-falha ou todos os nodos em c_{i,s} estão falhos
10         se todos os nodos em c_{i,s} estão falhos então
11             apaga informações de diagnóstico do cluster
12         durma até o próximo intervalo de testes
13 para sempre
```

Algoritmo 1: Algoritmo *Hi-ADSD*.

Outros algoritmos de diagnóstico distribuído hierárquico já foram propostos, incluindo o algoritmo *Hi-ADSD with Detours* [Duarte Jr. et al. 2009]. Quando um nodo testa um nodo falho em um determinado *cluster*, o testador tenta obter informações sobre aquele *cluster* a partir de nodos de outros *clusters*, ao invés de executar mais testes. Esses caminhos alternativos são chamados *detours* e tem sempre tamanho máximo idêntico à distância no algoritmo original. No algoritmo *Hi-ADSD with Timestamps* após um nodo sem-falha ser testado, o testador obtém informações sobre um conjunto de $N/2$ nodos. Assim, um nodo obtém informações sobre um outro nodo através de vários caminhos. Para indicar qual informação é mais recente são usados *timestamps*, contadores de mudança de estados. No algoritmo *DiVHA (Distributed Virtual Hypercube Algorithm)* [Bona et al. 2008] os testes são assinalados aos nodos de forma que cada nodo tem apenas um testador em cada cluster. Para garantir esta unicidade, o algoritmo faz buscas e comparações no grafo local e determina quais os testes que cada nodo do sistema deve executar de acordo com a situação atual.

3. O Algoritmo VCube

Este capítulo descreve o algoritmo proposto por este trabalho, chamado de Hipercubo Virtual (*VCube*). O novo algoritmo representa uma nova estratégia de testes para o algoritmo *Hi-ADSD* que garante uma quantidade máxima de testes logarítmica mantendo o mesmo

limite para a latência. O restante deste capítulo está organizado da forma a seguir. A Seção 3.1 especifica e descreve o novo algoritmo. A Seção 3.2 exibe uma nova definição para a função $c_{i,s}$ utilizada no novo algoritmo. Na Seção 3.3 são desenvolvidas as provas necessárias para mostrar que o número de testes é no máximo $N \log_2 N$ e que a latência é no máximo $\log_2^2 N$. Por fim, na Seção 3.4 resultados experimentais obtidos através de simulações são apresentados e comparados com o algoritmo *Hi-ADSD*.

3.1. O Algoritmo

Considere um sistema S que consiste em um conjunto de N nodos, $n_0, n_1, \ldots, n_{N-1}$. Alternativamente chamamos o nodo n_i de nodo i. Assumimos que o sistema é completamente conectado, ou seja, existe um canal de comunicação entre qualquer par de nodos. Cada nodo pode assumir um de dois estados, falho ou sem-falha. A combinação dos estados de todos os nodos do sistema constitui a situação de falha do mesmo. Os nodos realizam testes em outros nodos em intervalos de testes, e testes feitos por nodos sem-falha são sempre corretos.

No algoritmo *VCube* os nodos são agrupados em *clusters* para realização de testes. Os *clusters* são conjuntos de nodos cujo tamanho é sempre uma potência de 2. O sistema como um todo é um *cluster* de N nodos. O algoritmo *VCube* executado pelo nodo i é apresentado em pseudo código no Algoritmo 2. No primeiro intervalo de testes o nodo i realiza testes em nodos do *cluster* que contém um nodo ($c_{i,1}$), no segundo intervalo de testes em nodos do *cluster* que contém dois nodos ($c_{i,2}$), e assim por diante, até que os nodos do *cluster* que contém $2^{\log_2 N - 1}$, ou $N/2$ nodos ($c_{i,\log_2 N}$). A equação 2 define a função $c_{i,s}$ para todo $i \in \{0, 1, \ldots, N-1\}$ e $s \in \{1, 2, \ldots, \log_2 N\}$. A função $c_{i,s}$ retorna a lista ordenada de nodos que o nodo i poderá testar durante a rodada de testes s. Define-se o símbolo $\oplus$ como a operação ou exclusivo (*XOR*). A Seção 3.2 descreve detalhadamente o funcionamento desta função.

$$c_{i,s} = i \oplus 2^{s-1} \parallel c_{i \oplus 2^{s-1},k} \mid k = 1, 2, \ldots, s-1 \tag{2}$$

O novo algoritmo é baseado na seguinte regra: antes do nodo i executar um teste no nodo $j \in c_{i,s}$, ele verifica se ele é o primeiro nodo sem-falha em uma lista ordenada de nodos que podem testar o nodo j na rodada s. Depois que o nodo i testa o nodo $j \in c_{i,s}$ como sem-falha, ele obtém novas informações de diagnóstico do nodo j sobre todos os nodos do sistema. Este processo é repetido para todos os nodos pertencentes a $c_{i,s}$, finalizando uma rodada de testes.

Como um testador pode obter informações sobre um determinado nodo através de vários outros nodos, o algoritmo utiliza *timestamps*. Inicialmente todos os nodos são considerados sem-falha e o *timestamp* correspondente é zero. Quando um evento é detectado, isto é, um nodo sem-falha se torna falho ou vice-versa, o *timestamp* correspondente é incrementado. Assim, um *timestamp* par corresponde a um nodo sem-falha e um *timestamp* ímpar corresponde a um nodo falho.

O uso de *timestamps* torna o algoritmo capaz de diagnosticar eventos dinâmicos, ou seja, eventos que ocorrem antes do evento anterior ter sido completamente diagnosticado. Considerando este modelo de falhas dinâmicas é necessário especificar um procedimento para a recuperação dos nodos que são reparados durante a execução do algoritmo.

Utilizamos a mesma estratégia introduzida pelo algoritmo *Hi-ADSD with Timestamps*, onde um campo chamado *u-bit* é utilizado para representar se um valor de *timestamp* é válido ou não.

O nodo i mantém um vetor de *timestamps* $t_i[0..N-1]$. Após obter informação de um nodo j testado como sem-falha, o nodo i atualiza um *timestamp* local apenas quando o valor correspondente obtido for maior que o atual. Uma vantagem adicional desta abordagem é que a latência média do algoritmo é melhorada, pois testadores podem obter novas informações de diagnóstico sobre todos os nodos do sistema através de qualquer nodo testado.

Para evitar que o todo o vetor $t_j[]$ seja transferido quando o nodo i testa o nodo j como sem-falha, é possível implementar uma solução simples na qual o nodo j só envia novas informações, ou seja, informações que mudaram desde a última vez que o nodo j foi testado pelo nodo i.

É importante observar que o sistema é assíncrono, ou seja, em um certo momento, nodos diferentes do sistema podem estar testando *clusters* de tamanhos diferentes. Um nodo executando o algoritmo não é capaz de determinar quais testes estão sendo realizados por outros nodos do sistema em um dado momento. Mesmo se inicialmente os nodos estão sincronizados, após a falha e recuperação de alguns deles o sistema perderia sua sincronicidade. Este fato gera consequências no desempenho do algoritmo, que são tratadas na Seção 3.3.

```
1  Algoritmo VCube executado no nodo i
2  repita
3      para s ← 1 até log_2 N faça
4          para todo j ∈ c_{i,s} | i é o primeiro nodo sem falha ∈ c_{j,s} faça
5              teste(j)
6              se j está sem-falha então
7                  se t_i[j] mod 2 = 1 então  t_i[j] = t_i[j] + 1
8                  obtém informações de diagnóstico
9              senão
10                 se t_i[j] mod 2 = 0 então  t_i[j] = t_i[j] + 1
11         durma até o próximo intervalo de testes
12 para sempre
```

Algoritmo 2: Algoritmo *VCube*.

3.2. A Função $c_{i,s}$

A função $c_{i,s}$ é a base do algoritmo, pois ela é capaz de determinar para um nodo i quais são os *clusters* que ele deverá testar durante cada rodada de teste. A função $c_{i,s}$ retorna a lista ordenada de nodos que o nodo i poderá testar durante a rodada de testes s. Definimos o símbolo $\oplus$ como a operação ou exclusivo (*XOR*). A equação 3 mostra a definição da função $c_{i,s}$, e um exemplo de uso da função para um sistema com 8 nodos pode ser visto na Tabela 2.

$$c_{i,s} = i \oplus 2^{s-1} \parallel c_{i \oplus 2^{s-1},k} \mid k = 1, 2, \ldots, s-1 \tag{3}$$

Para entendermos melhor como a função $c_{i,s}$ funciona, primeiramente vamos lembrar que em um hipercubo existe uma aresta entre o nodo i e o nodo j se, e somente se, na representação binária de i e j existe apenas um *bit* diferente [LaForge et al. 2003]. Também devemos lembrar que dado um número i podemos inverter um de seus bits, utilizando a operação ou exclusivo (*XOR*), simplesmente fazendo $i \oplus 2^x$, onde x é a posição do bit que gostariamos de inverter e $\oplus$ é a operação ou exclusivo (*XOR*).

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

Tabela 2. $c_{i,s}$ para um sistema com 8 nodos.

A função $c_{i,s}$ mostrada na equação 3 é uma contribuição se comparada à versão original especificada pelo algoritmo *Hi-ADSD*. A ordem criada por ela tem propriedades importantes para o novo algoritmo, vistas a seguir.

A Tabela 3 mostra um exemplo da função $c_{i,s}$ calculada para um sistema com 8 nodos onde $s = 3$. Podemos ver que nenhum nodo é repetido nas colunas da tabela, ou seja, $\forall x, y \in \{0, 1, ..., N-1\}, k \in \{0, ..., 3\} \mid x \neq y : c_{x,3}[k] \neq c_{y,3}[k]$. Para provarmos este fato, utilizamos indução.

i	$c_{i,3}[0]$	$c_{i,3}[1]$	$c_{i,3}[2]$	$c_{i,3}[3]$
0	4	5	6	7
1	5	4	7	6
2	6	7	4	5
3	7	6	5	4
4	0	1	2	3
5	1	0	3	2
6	2	3	0	1
7	3	2	1	0

Tabela 3. Detalhe da $c_{i,3}$ para um sistema com 8 nodos.

Teorema 1: $\forall i, j, s, p \mid i \neq j : c_{i,s}[p] \neq c_{j,s}[p]$.

Prova: A prova será por indução em s.

Base: Considere $s = 1$. Para um nodo i a função $c_{i,1} = i \oplus 2^{s-1} = i \oplus 1$. Como 1 em binário tem apenas o *bit* menos significativo como 1, a operação *XOR* irá inverter o *bit* menos significativo de i. Considere i como sendo par, ou seja, seu *bit* menos significativo é 0: assim temos que $i \oplus 1 = i + 1$. Considere i como sendo ímpar, ou seja, seu *bit* menos significativo é 1: assim temos que $i \oplus 1 = i - 1$. Portanto, para qualquer $i, j \mid i \neq j$, $c_{i,1}[0] \neq c_{j,1}[0]$.

Hipótese de indução: Para todo $i, j, p \mid i \neq j : c_{i,k}[p] \neq c_{j,k}[p]$.

Passo: Considere $s = k + 1$.

Para um nodo i a função $c_{i,k+1} = i \oplus 2^k \parallel c_{i \oplus 2^k,1} \parallel c_{i \oplus 2^k,2} \parallel ... \parallel c_{i \oplus 2^k,k}$. Considere a operação $i \oplus 2^k$. Como 2^k tem apenas o k-ésimo *bit* como 1, a operação *XOR* irá inverter

o k-ésimo *bit* de i. Suponha que $j = i \oplus 2^k$, temos que $c_{j,k+1} = (i \oplus 2^k) \oplus 2^k, ...$ Temos então que $(i \oplus 2^k) \oplus 2^k = i$, ou seja, se $c_{i,k+1}[0] = j$ então $c_{j,k+1}[0] = i$.

Como $c_{i,k+1}[0]$ é único para cada i e pela hipótese $c_{i \oplus 2^k,1}$, $c_{i \oplus 2^k,2}$, ... , $c_{i \oplus 2^k,k}$ também são únicos, $c_{i,k+1}$ é único, e portanto, $c_{i,k+1}[p] \neq c_{j,k+1}[p], \forall j \neq i, p$.

Além da função $c_{i,s}$ representar quais nodos o nodo i pode testar na rodada de testes s, ela também representa para o nodo i durante a rodada de testes s, a ordem dos nodos que poderão testar o nodo i nesta rodada. Tecnicamente qualquer ordem pode ser utilizada para representar os nodos que podem testar um outro nodo em uma rodada de testes. Entretanto uma função que satisfaça o *Teorema 1* garante que a cada rodada de testes todos os nodos sem-falha irão realizar pelo menos um teste.

Considere, por exemplo, $c_{5,3} = \{1, 0, 3, 2\}$. Na terceira rodada de testes o nodo 5 irá verificar se ele é o primeiro nodo sem-falha nas listas ordenadas $c_{1,3}$, $c_{0,3}$, $c_{3,3}$ e $c_{2,3}$, respectivamente, e realizar o teste em caso positivo. É garantido que o nodo 5 irá realizar pelo menos um destes testes pois em alguma destas ordens ele será o primeiro. Por outro lado, os nodos 1, 0, 3 e 2, também na terceira rodada de testes, estando todos sem-falhas, irão verificar qual o primeiro nodo sem-falha em $c_{5,3}$, fazendo com que apenas um deles teste o nodo 5.

O uso de uma função $c_{i,s}$ que satisfaz o *Teorema 1* permite, de uma certa forma, manter as propriedades logarítmicas mesmo quando nodos se tornam falhos. A nova função $c_{i,s}$ mantém uma ordem progressiva de acordo com o hipercubo, fazendo com que as arestas criadas por nodos falhos conectem os nodos sem-falha de uma maneira mais similar ao hipercubo original. Suponha que em uma função $c_{i,s}$ o valor de $c_{i,3}$ fosse igual a $\{4, 5, 6, 7\}$ para todo $i \in \{0, 1, 2, 3\}$, ou seja, caso o nodo 4 esteja sem-falha ele será responsável por testar os nodos 0, 1, 2 e 3. Dependendo da função $c_{i,s}$ a carga de testes sobre um nodo pode chegar a ordem de N, e poucas falhas podem fazer com que o desempenho do algoritmo seja prejudicado consideravelmente. A função $c_{i,s}$ proposta distribui a responsabilidade pelos testes de forma igualitária entre todos os nodos.

3.3. Provas Formais

Nesta seção são apresentadas as provas de corretude e pior caso da latência de diagnóstico do algoritmo. Para estas provas assumimos que a situação de falhas do sistema não muda por um período suficiente de tempo. Vale lembrar que as provas de corretude dos algoritmos *Hi-ADSD* e *Adaptive-DSD* também consideram esta asserção.

Definição 1: *O grafo $T(S)$ é um grafo direcionado cujos vértices são os nodos de S. Para cada nodo i, e para cada cluster $c_{i,s}$, existe uma aresta de i para j se i testou j como sem-falha na rodada de testes mais recente em que i testou o cluster $c_{i,s}$.*

Lema 1: *Para qualquer nodo i, qualquer s, e a qualquer instante de tempo t_i, são necessárias, no máximo, $\log_2 N$ rodadas de testes para que o nodo i teste o cluster $c_{i,s}$.*

Prova: O lema segue da definição do algoritmo, ou seja, em um determinado intervalo de testes o nodo i testa pelo menos um nodo de $c_{i,s}$. Como o nodo i testa

$\log_2 N$ *clusters*, são necessárias $\log_2 N$ rodadas de testes para que todos os *clusters* sejam testados. Portanto, no pior caso, para t_i imediatamente após o teste de um dado *cluster*, poderá levar até $\log_2 N$ rodadas de testes para que este *cluster* seja testando novamente.

Teorema 2: *O menor caminho entre qualquer par de nodos sem-falha em $T(S)$ contém, no máximo, $\log_2 N$ arestas.*

Prova: A prova será por indução em t, para um sistema de 2^t nodos.

Base: Considere um sistema de 2^1 nodos. Cada nodo testa o outro, portanto o menor caminho entre eles em $T(S)$ contém uma aresta.

Hipótese de indução: Assuma que para um sistema com 2^k nodos o menor caminho entre qualquer par de nodos em $T(S)$ contém, no máximo, k arestas.

Passo: Por definição, um sistema de 2^{k+1} nodos é composto por dois *clusters* de 2^k nodos. Considere o subgrafo de $T(S)$ que contém apenas os nodos de um destes *clusters*. Pela hipótese anterior o menor caminho entre qualquer par de nodos neste subgrafo contém, no máximo, k arestas. Considere quaisquer dois nodos i e j. Se i e j estão no mesmo *cluster* de 2^k nodos, o menor caminho entre eles em $T(S)$ contém, no máximo, k arestas. Agora considere o caso em que i e j estão em *clusters* de tamanho 2^k diferentes. Sem perda de generalidade, considere o menor caminho entre i e j. O nodo j recebe um teste de algum nodo do *cluster* ao qual o nodo i pertence: chame este nodo de p. Em $T(S)$, a menor distância entre p e j é de uma aresta, e a menor distância de i até p é, no máximo, k arestas. Portanto, o menor caminho de i a j contém, no máximo, $k+1$ arestas.

Teorema 3: *Considere uma situação de falha em um dado momento. Após, no máximo, $\log_2^2 N$ rodadas de testes, cada nodo que permaneceu sem-falha por este período corretamente determina aquela situação de falha.*

Prova: Foi provado no Teorema 2 que o menor caminho entre qualquer par de nodos em $T(S)$ tem, no máximo, $\log_2 N$ arestas. Segundo o Lema 1, cada um dos testes correspondentes a uma aresta em $T(S)$ pode levar até $\log_2 N$ rodadas de testes para ser executado no pior caso. Então, temos até $\log_2 N$ testes diferentes para serem executados, e cada um deles pode levar até $\log_2 N$ rodadas de testes para ser executado. Então, no total, estes testes podem levar no máximo $\log_2 N * \log_2 N$ rodadas de testes para serem executados. Portanto, podem ser necessárias até $\log_2^2 N$ rodadas de teste para um nodo sem-falha obter informação de diagnóstico sobre um evento em S.

Teorema 4: *O número de testes executados por todos os nodos do sistema executando o algoritmo VCube é no máximo $N \log_2 N$ a cada $\log_2 N$ rodadas de testes.*

Prova: Considere os testes executados no nodo j. Para cada $c_{j,s} \mid s = 1, ..., \log_2 N$, o nodo j é testado pelo primeiro nodo sem-falha em $c_{j,s}$. Cada nodo sem-falha em $c_{j,s}$ utiliza a mesma lista ordenada de nodos que contém ele mesmo para verificar se ele é o primeiro nodo sem-falha ou não. Em caso afirmativo ele deve testar o nodo j. Assim cada nodo é testado no máximo $\log_2 N$ vezes a cada $\log_2 N$ rodadas de teste. Como o sistema tem N nodos, o número total de testes é no máximo $N \log_2 N$.

Deve ficar claro que, assim como nos algoritmos *Hi-ADSD* e *Adaptive-DSD*, o limite do número de nodos falhos para que os nodos sem-falha realizem um diagnóstico consistente é $N - 1$. Neste caso, se $N - 1$ nodos estão falhos, o nodo sem-falha irá testar

todos os nodos para diagnosticar o sistema.

Também devemos notar que o número de nodos do sistema, N, não necessariamente precisa ser uma potência de 2. Neste caso, os nodos testadores devem ignorar os $2^{\lceil \log_2 N \rceil} - N$ nodos inexistentes durante testes e recuperação de informações de diagnóstico.

3.4. Resultados Experimentais

Os resultados experimentais do algoritmo *VCube* foram obtidos tanto através de simulação, apresentados nesta seção, como em execução no Planet-Lab, apresentados no próximo capítulo. As simulações foram conduzidas utilizando a linguagem de simulação baseada em eventos discretos, SMPL [MacDougall 1987]. Nodos foram modelados como *facilities* SMPL e cada nodo foi identificado por um número de *token* SMPL. As falhas foram modeladas como um nodo sendo reservado, que é apenas um estado interno da ferramenta de simulação. Durante cada teste, o estado do nodo é verificado, e se este está reservado ele é considerado falho. Caso contrário o nodo é considerado sem-falha. Cada nodo inicia seus testes em um *cluster* escolhido aleatoriamente, fazendo com que os nodos do sistema não estejam completamente sincronizados.

Todos os experimentos foram executados em uma rede de 512 nodos. Os resultados mostrados são a latência média e a quantidade de testes média. A latência é o número de rodadas de testes necessárias para que todos os nodos sem-falha detectem um evento. A quantidade de testes mostrada representa o número de testes realizados em média no sistema em uma situação estável, ou seja, a quantidade de testes após todos os nodos diagnosticarem o evento em questão durante $\log_2 N$ rodadas de testes. A simulação leva em conta a asserção de que um evento ocorre apenas quando o evento anterior tiver sido completamente diagnosticado pelo sistema.

As Figuras 3 e 4 mostram a quantidade de testes e a latência média, respectivamente, em uma rede onde os nodos falham de uma maneira a simular o pior caso do algoritmo *Hi-ADSD*. Os nodos de identificadores 0 a 255 falham de maneira aleatória até que todos estejam falhos e em seguida o mesmo acontece com os nodos de identificadores 256 a 511. Assim, com $N/2$ nodos falhos no mesmo *cluster*, comparamos o comportamento do *VCube* com o *Hi-ADSD*.

Pode-se observar que o *Hi-ADSD* tem um comportamento quadrático quando o número de nodos falhos se aproxima de $N/2$, chegando ao seu pior caso quando o número de testes chega a $N^2/4 = 65536$. Durante os mesmos casos de testes, o algoritmo *VCube* se comporta de maneira logarítmica, nunca passando de seu limite de $N \log_2 N = 4608$ testes por $\log_2 N$ rodadas de testes. Analisando a latência média que ambos algoritmos apresentaram, podemos ver que a nova estratégia de testes apresenta uma diferença considerável se comparada à estratégia original, apesar de ambas apresentarem uma latência máxima teórica de $\log_2^2 N$ rodadas de testes. Esta diferença se deve ao fato de que o algoritmo *Hi-ADSD* original não utiliza *timestamps*, que diminuem consideravelmente a latência.

As Figuras 5 e 6 mostram a quantidade de testes e a latência média, respectivamente, em uma rede onde os nodos falham de uma maneira aleatória. Novamente podemos ver que a latência média do *VCube* é mais baixa que a do *Hi-ADSD*. Um resultado a ser destacado é que, na média, o número de testes executados pelos dois algoritmos é

muito parecido. Estes resultados foram obtidos a partir da execução de cerca de 7000 simulações de cenários de falhas para os dois algoritmos, porém existem $511!$ cenários de falhas possíveis. Como os nodos que irão falhar são escolhidos aleatoriamente, um caso como o pior caso do algoritmo *Hi-ADSD* tem uma probabilidade muito pequena de acontecer.

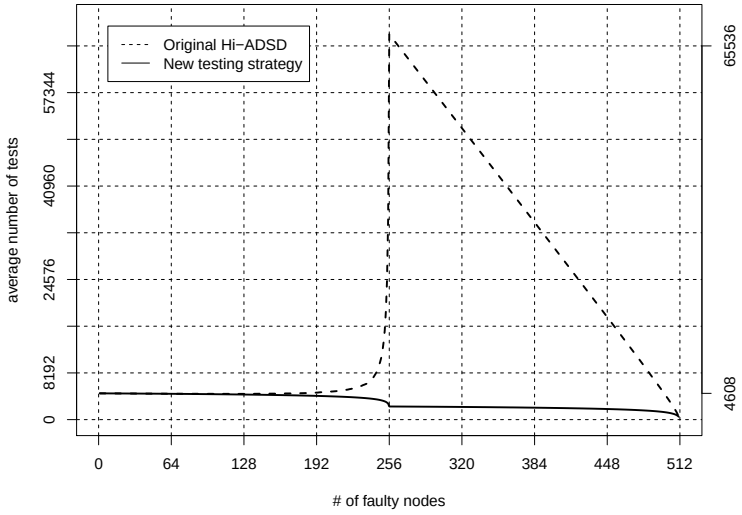


Figura 3. Número de testes médio em uma rede estável de 512 nodos no pior caso do algoritmo *Hi-ADSD*.

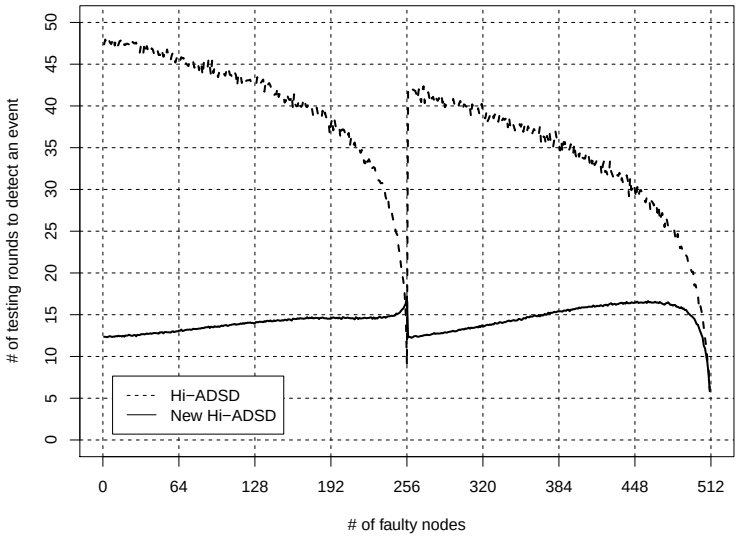


Figura 4. Latência média para detecção de um evento em uma rede de 512 nodos no pior caso do algoritmo *Hi-ADSD*.

4. Conclusão

Este trabalho apresentou o novo algoritmo *VCube* para o diagnóstico distribuído hierárquico que garante propriedades logarítmicas tanto para sua latência quanto para a quantidade máxima de testes executados. Em comparação com outro algoritmo de diagnóstico distribuído hierárquico, o *Hi-ADSD*, o algoritmo proposto neste trabalho reduz

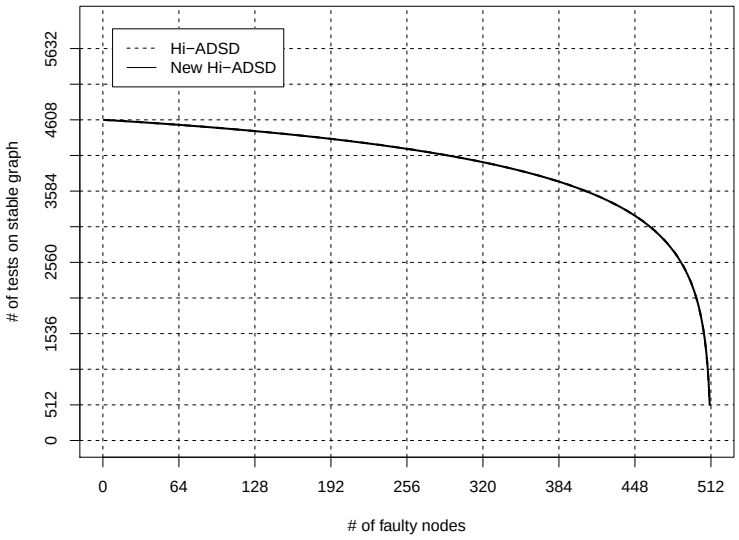


Figura 5. Número de testes médio em uma rede estável de 512 nodos.

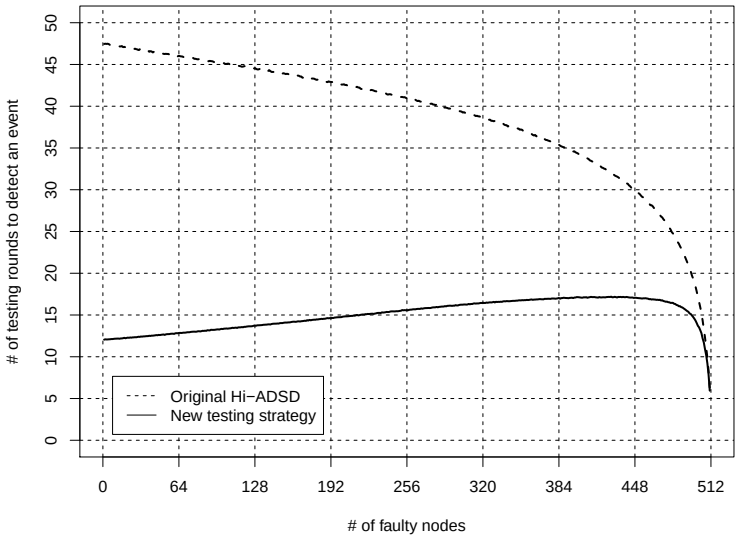


Figura 6. Latência média para detecção de um evento em uma rede de 512 nodos.

o número máximo de testes executados, mantendo o mesmo pior caso para a latência. Enquanto no *Hi-ADSD* o número de testes executados no pior caso é quadrático, no algoritmo proposto o limite máximo é de $N \log_2 N$ testes a cada $\log_2 N$ rodadas de testes. Ambos os algoritmos têm latência máxima de $\log_2^2 N$ rodadas.

O algoritmo *VCube* organiza a execução de testes em *clusters* progressivamente maiores. Na estratégia de testes proposta os testes são realizados com base na seguinte regra: um nodo i na rodada de testes s deve testar o nodo $j \in c_{i,s}$ se, e somente se, o nodo i é o primeiro nodo sem-falha em uma lista ordenada de nodos que podem testar o nodo j na rodada de teste s. Quanto um nodo sem-falha é testado informações de diagnóstico sobre todo o sistema são enviadas. Uma nova função $c_{i,s}$ foi apresentada para representar tanto os nodos que o nodo i pode testar na rodada de testes s quanto os nodos que podem testar o nodo i nesta mesma rodada.

Resultados de simulação inclusive comparando o novo algoritmo com o algoritmo *Hi-ADSD* foram apresentados. Com relação ao número de testes o novo algoritmo apresentou uma quantidade logarítmica de testes em cenários onde o algoritmo *Hi-ADSD* apresentou comportamento quadrático. A latência média do novo algoritmo foi inferior ao algoritmo *Hi-ADSD* em todos os cenários, devido tanto à nova estratégia de testes quanto à obtenção de informações de diagnóstico utilizando *timestamps*.

Trabalhos futuros incluem trabalhar em uma especificação do algoritmo que permite que cada nodo execute testes nos $\log_2 N$ clusters em uma única rodada, mantendo um custo razoável (logarítmico) e, ao mesmo tempo, baixando a latência máxima para $\log_2 N$ rodadas de teste ao invés de $\log_2^2 N$ no pior caso. Reduzir a computação necessária para a execução do algoritmo também deve ser considerada para sua execução em redes com um número grande de nodos. Diversas otimizações são possíveis para evitar que a toda rodada de testes os testadores executem múltiplas verificações para determinar exatamente quais testes devem realizar. Outra linha de trabalho pode expandir a definição do algoritmo para que seja possível realizar diagnóstico em redes de topologia arbitrária.

Referências

Bona, L., Fonseca, K., and Duarte Jr., E. P. (2008). Hyperbone: A scalable overlay network based on a virtual hypercube. In *Network Operations and Management Symposium, 2008. NOMS 2008. IEEE*, pages 1025–1030.

Duarte Jr., E., Ziwich, R., and Albini, L. (2011). A survey of comparison-based system-level diagnosis. *ACM Computing Surveys*.

Duarte Jr., E. P., Albini, L., Brawerman, A., and Guedes, A. (2009). A hierarquical distributed fault diagnosis algorithm based on clusters with detours. In *Network Operations and Management Symposium, 2009. LANOMS 2009. Latin American*, pages 1–6.

Duarte Jr., E. P. and Nanya, T. (1998). A Hierarquical Adaptive Distributed System-Level Diagnosis Algotithm. *IEEE Transactions on Computers.*, 47(1):34–45.

Hakimi, S. L. and Nakajima, K. (1984). On Adaptive System Diagnosis. *IEEE Transactions on Computers.*, C-33(3):234–240.

Hosseini, S. H., Kuhl, J. G., and Reddy, S. M. (1984). A Diagnosis Algorithm for Distributed Computing Systems with Dynamic Failure and Repair. *IEEE Transactions on Computers.*, C-33(3):223–233.

LaForge, L., Korver, K., and Fadali, M. (2003). What designers of bus and network architectures should know about hypercubes. *Computers, IEEE Transactions on*, 52(4):525–544.

MacDougall, M. (1987). *Simulating computer systems: techniques and tools*. MIT Press, Cambridge, MA, USA.

Masson, G., Blough, D., and Sullivan, G. (1996). Fault-tolerant computer system design. chapter System diagnosis, pages 478–536. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Preparata, F., Metze, G., and Chien, R. T. (1967). On the Connection Assignment Problem of Diagnosable Systems. *IEEE Transactions on Computers.*, 16:848–854.

A Test Process Model to Evaluate Performance Impact of Privacy Protection Solutions

Victor Mello, Tania Basso, Regina Moraes

Faculdade de Tecnologia – Universidade Estadual de Campinas (UNICAMP)
Caixa Postal 456 – 13484-332 - Campus I – Limeira – SP – Brasil

victorleonel@gmail.com, {taniabasso,regina}@ft.unicamp.br

Abstract. *Organizational Information Systems (IS) collect, store, and manage personal information through web applications and services. Due to regulation laws and to protect the privacy of clients, such information must be kept private. Some solutions were developed to protect privacy personal information. Obviously, this additional resource will produce a performance impact and evaluating it is essential to determine the feasibility of the solution. This paper presents a process model to evaluate the performance impact introduced by privacy protection solutions in web applications. Case study shows the tests were useful to identify the conditions in which the solution under evaluation is able to work with minimal performance impact.*

1. Introduction

Nowadays, applications and services provided via web such as e-commerce, banking and financial services are essential and widely used by society. The provision of these services, performed with the aid of a browser, is only possible due to the support of computers network that allows communication between suppliers and users. Most of time, to use these services, users and customers need to provide personal information. Once sent across the network, the information has become known by the service provider and, often, by other business partners that not even had any interaction or involvement with the users. It means that once personal information is made available they are no longer under control of their owner in respect to how they are used and their consequences. This finding raises privacy concerns.

The need to ensure the security and privacy of personal information handled by web services and applications has led to a significant development of technology. This need arises from regulations laws (the companies and organizations that hold private data have the obligation and responsibility to protect them) and competitive differentials (the more a company protect the privacy of its customers and business partners, the better is its reputation). The privacy policy is a resource often used in this context.

The description of organization's practices on information collection, use, and disclosure is the focus of the internet privacy policies. These privacy policies intend to protect the organization and to signal integrity commitment to site visitors. To guide browsing and transaction decisions, consumers use the stated website policies (Earp *et al.* 2005). They are so important that can influence the organization's credibility: if the privacy policies are clearly and explicitly stated, then the visitor/consumer perceives the organization as more trustworthy (Han and Maclaurin 2002).

Besides the privacy policies definition, mechanisms to enforce them are necessary to make sure companies keep their privacy promises to consumers and business partners. A well-known approach for protecting privacy in scenarios based on web applications and services is to manage data using access control resources, that can be defined as "*a set of rules by which human users, or their representatives, are authenticated and by which access by these users to applications and other services information is granted or denied*" (IETF 2013). Thus, the information access control mechanism must be embedded into privacy-enhancing technologies. Obviously, adding access control to the web application or service will produce a performance impact. It is desired that this impact be small comparing to its nonuse. Evaluating this performance impact (through performance tests) is essential to determine the feasibility of using the privacy solution, as well as ensuring it is appropriate when deployed and used.

To evaluate the performance impact a test process model is required. This process model must support the definition of tests scenarios, their execution and the comparison of the tests results, especially when using and not using the privacy solution. Also, the process allows evaluating the solution's feasibility in different scenarios and different situations of using the web application or service (as, for example, with different amount of records in the database or different amount of simultaneous access by different users). This paper proposes a test process model to evaluate the performance impact of using privacy protection solutions in web applications.

The test process model was applied to the evaluation of a privacy solution integrated in a web application that represents an online book store. Results show that the tests were useful to identify in what conditions the solution under evaluation is able to work with minimal performance impact. Also, it helps to evaluate the scalability in terms of the number of registers in the database (the more registers, the more performance impact). This suggests the situations in which the privacy solution can be used as a simple and effective alternative to privacy protection.

The paper is organized as follows. Section 2 introduces the background and related work. Section 3 contextualizes the privacy protection issue and presents the privacy solution evaluated in this paper. Section 4 presents the proposed test process model. Section 5 shows the case study and the results and discussions. Finally, Section 6 shows the conclusions and future works.

2. Background and Related Work

The system quality is assured by the test activities, which main goal is to find the system failures or to validate the non-functional requirements of a system. Among different types of system test (Myers 1979), the performance test is gaining ground in the Web environments. The main goal of performance testing is to determine if the system performance reaches the restrictions imposed by the owners related to the response average time and throughput, in accordance with the system requirements (Pressman 2009).

Molinari (2003) presented the relationship between test and quality requirements in Web applications and highlights the quality requirements that most compromise the correct operation of these systems if they are not properly tested. In accordance with

Molinari, performance is the quality requirement that should demand more accurate test in Web applications.

The work most close related to ours was presented by Torres-Zenteno et al. (2006), which proposed a testing process model to guide the experiments on a GIS (Geographic Information System) Web application. In that work, the experiments were focused on the more critical use cases of the WebMaps system that helps the users to plan the agricultural distribution in some regions of interest. The performance tests are indispensable in the context of that system to ensure its applicability in its operational phase. The proposed process deals with the specific characteristics of the SIG Web applications, i.e., large volume of data and concurrent users. Our work deals with a completely different application that is an e-commerce Web application based on TPC-W standard (TPC 2014).

The work presented by Basso et al. (2013) proposes a policy model for data privacy protection and a mechanism as a protection layer, which is independent of the web application. The mechanism aims to enforce different policies derived from the model. The solution allows users to define the policies without requiring specific or in-depth knowledge about the web application, even expressing their preferences about each one of their personal information. However, although the model predicts the user preferences in detail, the protection layer enforces the access control in the web application code, leaving the data more vulnerable to attack, once they leave the database in clear. Moreover, there is no process to guide the experiments that were executed in an ad-hoc manner. This paper improves the work presented before, testing a new database framework that was implemented in the database layer, reducing the solution vulnerability. The framework encapsulates the mechanism to express and to enforce polices and it is able to guarantee that the personal privacy preferences will be fulfilled as before. Moreover, a testing process model was defined and was used to guide the experiments.

3. Privacy Protection

Although privacy can be defined in many ways, we use to this work the definition of Bertino et al. (2008), which says that "*privacy is the right of an entity to be secure from unauthorized disclosure of sensible information that are contained in an electronic repository*".The increasing use of personal information in Internet-based applications has raised privacy concerns over the world. Due to the fast advancement of information technology and the Internet, the users have a very low level of control of their information after they are released on the web. Companies and organizations are investing on protection of its customer's personal data privacy as a competitive differential. Also, to support and encourage the privacy protection, governments of different countries are investing in laws and regulations. To contextualize the current scenario of privacy protection regarding the collection and use of personal information, as well as laws that govern privacy on the Internet, some issues are described below.

The value of personal information. Usually, medium or big companies use sophisticated technologies to obtain personal information of their customers or potential customers. This personal information is valuable from the market viewpoint because it allows constructing consumer profiles and, thus, adopting strategies such as

advertisements targeted to their interest profiles. Selling such information to third parties is also common.

Some researches show that, although people approve the collection of their personal information (with their permission and purposes which they agree), they do not trust in many companies and resent when any of them is not clear on their plan to use personal information, or use such information for other purposes (Perkins and Markel 2004). For them, these actions constitute a privacy violation (Reay et al. 2009).

The Truste enterprise (Truste 2013) promoted a research in 2013 to assess and understand the main concerns of the costumers, the costumers trustworthy and business impact related to the online privacy of adults in the United States. More than two thousands internet users were interviewed and 89% revealed to be concerned about the privacy of your information when using the internet. In fact, the users concerns are justified: in a research performed by Accenture (2009), when 5500 business managers in 19 different countries were interviewed, 58% admitted that their enterprises had lost personal information from their customers and clients. Among them, 19% admitted that this event had occurred more than 5 times. Furthermore, 55% of the companies admitted that they had put the personal information available for third-parties or even outsourced the collection or processing of such information.

Privacy as a business competitive advantage. Based on the above reports, there is no doubt that investing in assuring privacy leads companies and organizations achieve a competitive advantage. Perkins and Markel (2004) affirm through a pragmatic argument, that ironically, companies should provide a rigorous protection of personal information of their consumers not only because it is the right thing to do, but for the reason that this protection is part of their larger interests. Innocuous privacy policies are suitable to a short-term business and robust privacy policies are suitable to a long-term business. So, companies and organizations should adopt robust privacy policies because people, in general, hate innocuous privacy policies (Perkins and Markel 2004).

In order to assess the impact of privacy concerns on the result of the use of online advertisements the Ponemom Institute (Ponemon 2010) interviewed, in 2010, 90 organizations that use the internet advertising as the main channel of communication in the United States. More than 70% of them agree that online advertising increases the organization marketing and their sales performance. Furthermore, 98% of the interviewed companies affirm that the privacy concerns limit their investments in online advertising.

Although there is interest by companies and organizations as well as tools and technologies to assist in privacy guarantee, government intervention in the form of laws is needed because the threats are fairly broad, malicious users are many, knowledge levels of the enterprises and organizations are very unequal, and in practice, privacy policies provided by these companies may vary and even may present ethical failures (Cate 2009, Perkins and Markel 2004).

Laws and regulations in different countries. Legislative approaches differ from country to country. Many countries have regulations for privacy information, such as China, India, the countries of the European Union (EU), United States, Canada, among others (Glass and Gresko 2012). Especially when one is dealing with

international business interactions, it is important to know the relevant privacy legislation. The two or more laws and relevant Statutes in several papers in the literature are mainly the European Union and the United States. In Brazil, according to Lemos (2013), there are no specific laws addressing the privacy of information on the Internet in Brazil. However, the Ministry of Justice has been working on drafting a law to protect personal data. The project, entitled "*Marco Civil da Internet*", was inspired by the model of the European Union and is under discussion in Brazilian Congress and still waits for vote (R7 Notícias 2014). A discussion on effectiveness of this project can be found in the article by Torres (2012). A text version of this law, dated from November 2012, which was delivered to the plenary, is available in Convergência Digital (2012).

Some cases of privacy violation. A recent serious example of privacy violation is the case of PRISM, a monitoring program created by the National Security Agency of the United States (NSA), in partnership with the FBI and with most technology companies (Google, Facebook, Microsoft, etc.). The PRISM has the proposal to monitor potential valuable international communications, which could pass by the companies' servers in the United States. With this, U.S. intelligence consulted audios, videos, photos, contents of emails, downloaded files and users' connections, violating their privacy (Veja 2013). The case had a great impact because it allowed spying information inclusive of presidents of several countries.

There are many other cases of privacy violation, as for example, when Google and other giants of the network were able to track users activities through automatic writing cookies in Safari browser used on Iphones, without the knowledge of users (Globonews 2012). And when a law student in Vienna noticed that all his deleted information in Facebook were not really deleted from the social network, making a group of Austrian students protest to press the Facebook to have privacy rules more severe (Youtube 2012).

The cases of privacy violation show that the way the companies and organizations protect and use personal information requires, in addition to ethics, systems and mechanisms for efficient privacy protection. Solutions for defining privacy policies and enforcing them have been proposed. Their feasibility should be evaluated.

3.1. A Database Framework for Expressing and Enforcing Personal Privacy Preferences

This section presents the privacy protection solution we selected to be used in our case study. It is a database framework, designed by Basso et al. (2014), for users to express their preferences about the privacy of their personal information. This selection was done due to the simplicity, ease of use and availability of the solution (it is free). Also, the flexible way it treats the personal privacy preferences is an important resource and is of our interest to evaluate its performance. The solution is described below.

In a broad view, the solution consists of a framework with a set of independent tables that can be added to the application's database. These tables will contain the necessary information to perform the access control according to predefined privacy policies, which consider users preferences about each personal information to be collected, stored and managed. The access control is performed by implemented database packages and has the advantage of filtering the data directly in the database,

contributing to security against possible attacks to the web application. Figure 1 shows a general view of the solution.

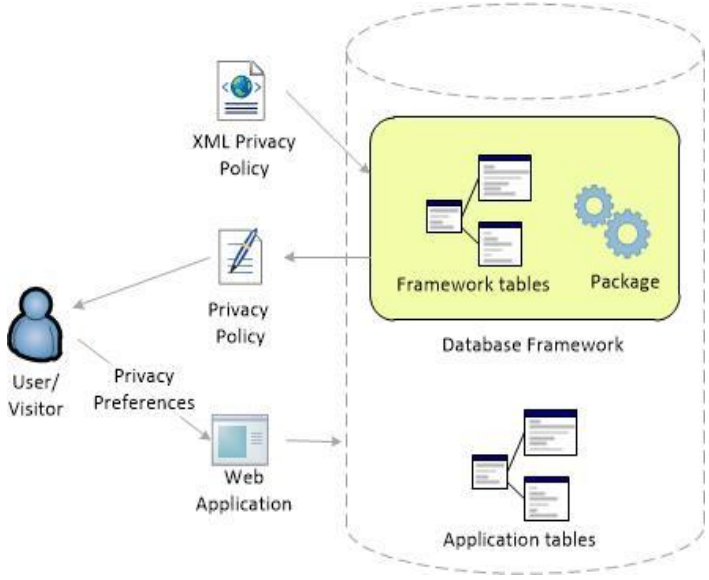


Figure 1. General view of the database framework solution (Basso et al. 2014).

In Figure 1, privacy policies are defined through XML files. These files are based on the policy model from Basso et al. (2013). These policies contain information about the system profiles and the personal data (associated with criticality levels) each profile can access or modify. These criticality levels are a scale of values to define how the information can be protected and will be further explained. A job (i.e., a combination of a schedule and a program, along with any additional arguments required by the program) is executed periodically to verify, automatically, in a specific application directory, the input of new policies. Then, the job maps these policies to the set of framework's tables.

After defining privacy policies, the normal process used by most companies is to present them to users and visitors, to inform them about company's privacy rules. Then, the user or visitor expresses their privacy preferences through the web application and this information are mapped in the both application and framework tables. These privacy preferences are expressed using predefined criticality levels. These criticality levels represent data with different requirements in terms of security, ranging from non-critical data to data that has to be extremely protected against unauthorized access. In order to identify the different levels of criticality and, consequently, how the data has to be protected, we established, for our study, the 5 levels described in the work of Vieira and Madeira (2005). They range from level 1 (non-critical data, i.e., data that does not represent any confidential information.) to level 5 (critical data that has to be exceptionally protected against unauthorized read and modification, using cryptography and auditing) (see Vieira and Madeira (2005) for more details). However, companies and organizations can establish their own levels according to their necessities.

Still in Figure 1, to guarantee the enforcement of privacy policies and, consequently, to respect users privacy preferences, a mechanism was developed and

integrated to the framework. This mechanism is an algorithm integrated to the infrastructure of a relational database system to enforce disclosure control. It provides constructs that allow masking personal information according to the privacy policies and users preferences. The mechanism was implemented using database packages, which are resources to encapsulate related procedures, functions, associated cursors and variables together as a unit in the database.

To better understand the potential and feasibility of the proposed solution in terms of performance impact, performance tests are required. Thus, a test process model was defined to help this activity. It is described in the next section.

4. The Test Process Model

This section describes the test process model that is proposed to guide performance tests when testing the introduction of privacy solutions in web systems. The process is based most on the works of Torres-Zenteno et al. (2006), which consists of a testing process model to guide the experiments on a GIS Web application, and Basso et al. (2013), which proposes a privacy solution integrated in the application layer and performs some performance tests in a not formalized way. Figure 2 shows the proposed test process model, using the notation of UML activity diagram.

In Figure 2, the process starts with the activity of planning environment. It is necessary to establish the tools and resources to be used in the tests. It is important that the test environment reproduces the application environment accurately. Then, the environment must be configured.

The next step in the process refers to choose the number of records to be inserted in the database. This step is important because the number of records can impact the performance of the web application under test. Tests can be done with different amount of records to evaluate the scalability of the privacy solution. Then, it is necessary to populate the database with predefined number of records.

Still in Figure 2, the activity of planning the performance test refers to the identification of the use cases that will be tested. The identification of critical use cases related to privacy protection is vital for achieving a good performance test. Planning the tests is followed by the creation of scenarios. It is important to note that one or more scenarios can be defined. After creating the scenarios, the virtual web users, which represent the concurrent users (or threads), are defined.

To run the tests scenarios, there is a decision activity. As we want to evaluate the performance impact introduced by the privacy solution, tests without the solution are required in order to make a comparison. So, tests can follow two paths: without enabling the privacy solution (NO path) or enabling the privacy solution (YES path). When choosing enabling the privacy solution, an activity to this procedure is represented in Figure 2 as setUseFilter = true. For both paths, the tests scenarios are run and the results are collected and analyzed independently. To evaluate the performance impact, these results must be compared (they should have been executed under the same conditions). Graphics can be created to help the evaluation.

The last activity in the test process model represents the consolidation of the performance tests, where the performance impact can be evaluated and, consequently, the feasibility of using the privacy protection solution.

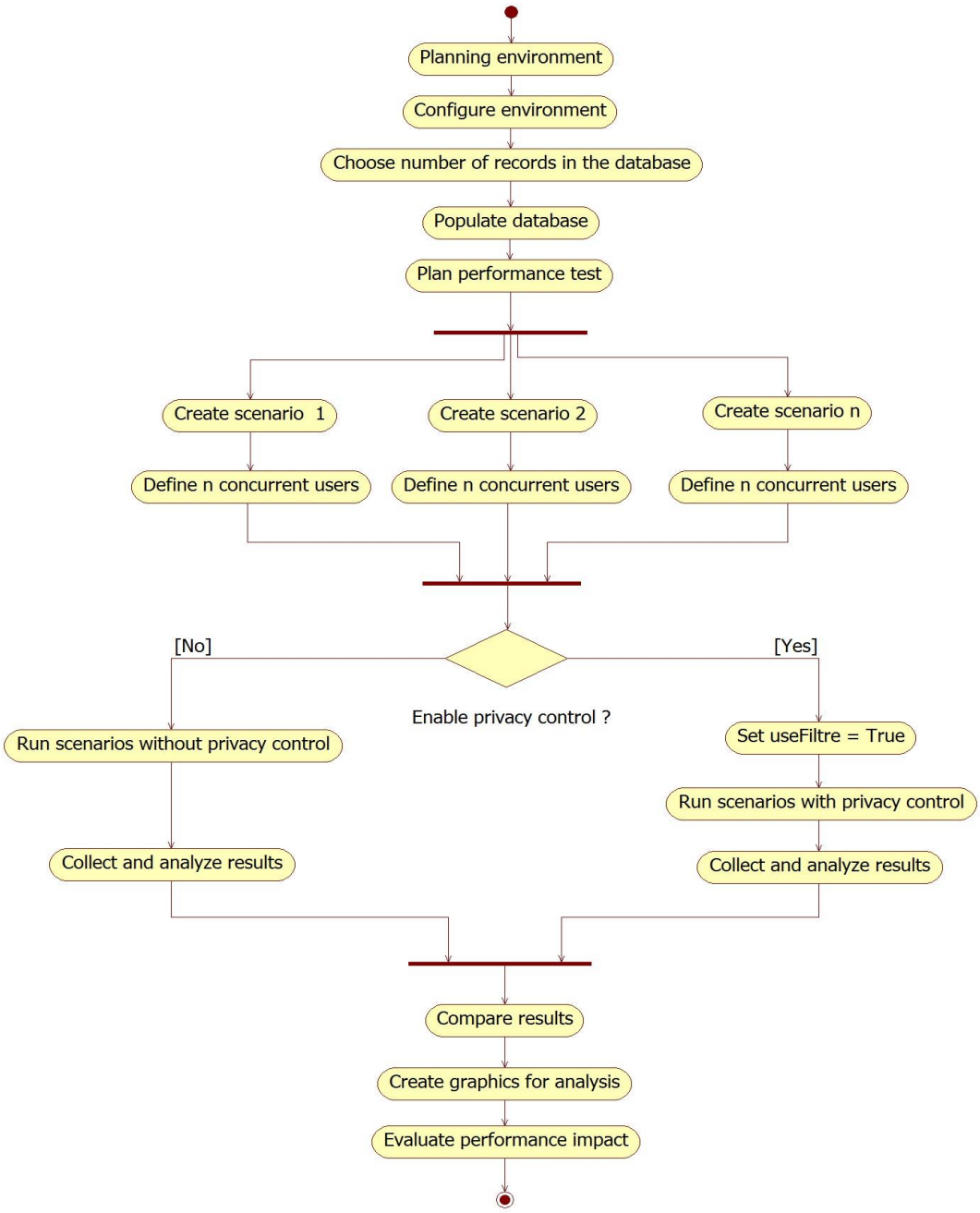


Figure 2. Proposed test process model to evaluate the performance impact of privacy protection solutions in web applications

5. Case Study

To better understand the potential of the selected privacy solution (Basso et al. 2014) and the applicability of our proposed test process model, a case study was performed.

The case study evaluates the scalability and performance impact of the proposed solution and also the correctness of the proposed process. Scalability is expressed in terms of the number of records being processed by the solution, i.e., the goal is to understand how much the number of records in the database application affects the performance. The performance impact can determine advantages or disadvantages of using the solution. For these experiments the performance was characterized by the average response time and throughput.

5.1 . Experimental Setup

The experiments were performed using an Intel Core 2 duo processor, 2 Gb RAM, Windows 7 Professional (32 bits) operating system. The same notebook houses a NetBeans IDE 7.3 and Tomcat 6.0.36. All the resources needs were locally installed for preliminary testing.

The web application used in the experiments is a Java implementation of a TPC-W (TPC 2014), which is a benchmark for web-based transactional systems where several clients access the website to browse, search, and process orders. To this study, the TPC-W implementation simulates a retail online book store. The components of the TPC-W database are defined to consist of a minimum of eight separate and individual base tables. The database engine used in the experiments is Oracle Database 11g Express Edition Release 11.2.0 (Oracle 2014) and the metrics were collected using the JMeter tool version 2.9 (Jmeter 2014).

The application scenario to perform the tests simulates a third-party user, with guest profile, trying to obtain data of a registered customer through a search process. To test including the privacy solution, privacy policies were implemented and the criticality level of each data of each customer was randomly generated through a database script.

For our experimental evaluation it was used, respectively, 500, 5,000, 50,000 and 500,000 records of customers in the database. These records emulate real typical online shopping data. The simulations of threads, that simulate concurrent users connected to the server application, ranged from 1 to 256 users for each set of records. Also, in order to understand the performance impact, the tests were performed without the privacy solution, to obtain baseline indicators. For each run of the experiment, the whole system is returned to its initial state in order to avoid cached data.

5.2. Overall Results Analysis

First of all, the average time was collected to understand the performance degradation due to the amount of data in the Database and the concurrent users (also called threads) of the web application. Figure 3 presents the overall results of the study, showing the average processing time of all requests for the customer search scenario. Presented in Figures 3a, 3c, 3e, the average time is given in milliseconds, representing from 1 up to 256 concurrent users. Also, Figures 3b, 3d and 3f show the throughput results for the same scenarios and amount of concurrent users. Throughput is calculated as the number of requests divided by unit of time. The time is calculated from the start of the first sample to the end of the last sample, including any intervals between them, as it is supposed to represent the load on the server.

Figures 3a, 3c and 3e shows the tests performed with, respectively, 500, 5,000 and 50,000 customers recorded in the database. The same for throughput: Figures 3b, 3d and 3f shows the tests performed with, respectively, 500, 5,000 and 50,000 customers. The use of different number of records can be necessary due to different quantities of information in the database for different companies or organizations sizes. It helps to evaluate the scalability of the privacy solution thinking about applying it in online commercial applications of small and midsize enterprises. Also, we have tried to perform tests with 500,000 customers, but it has exceeded the resources of the computational environment and the solution, making the analysis unfeasible and inaccurate for this case.

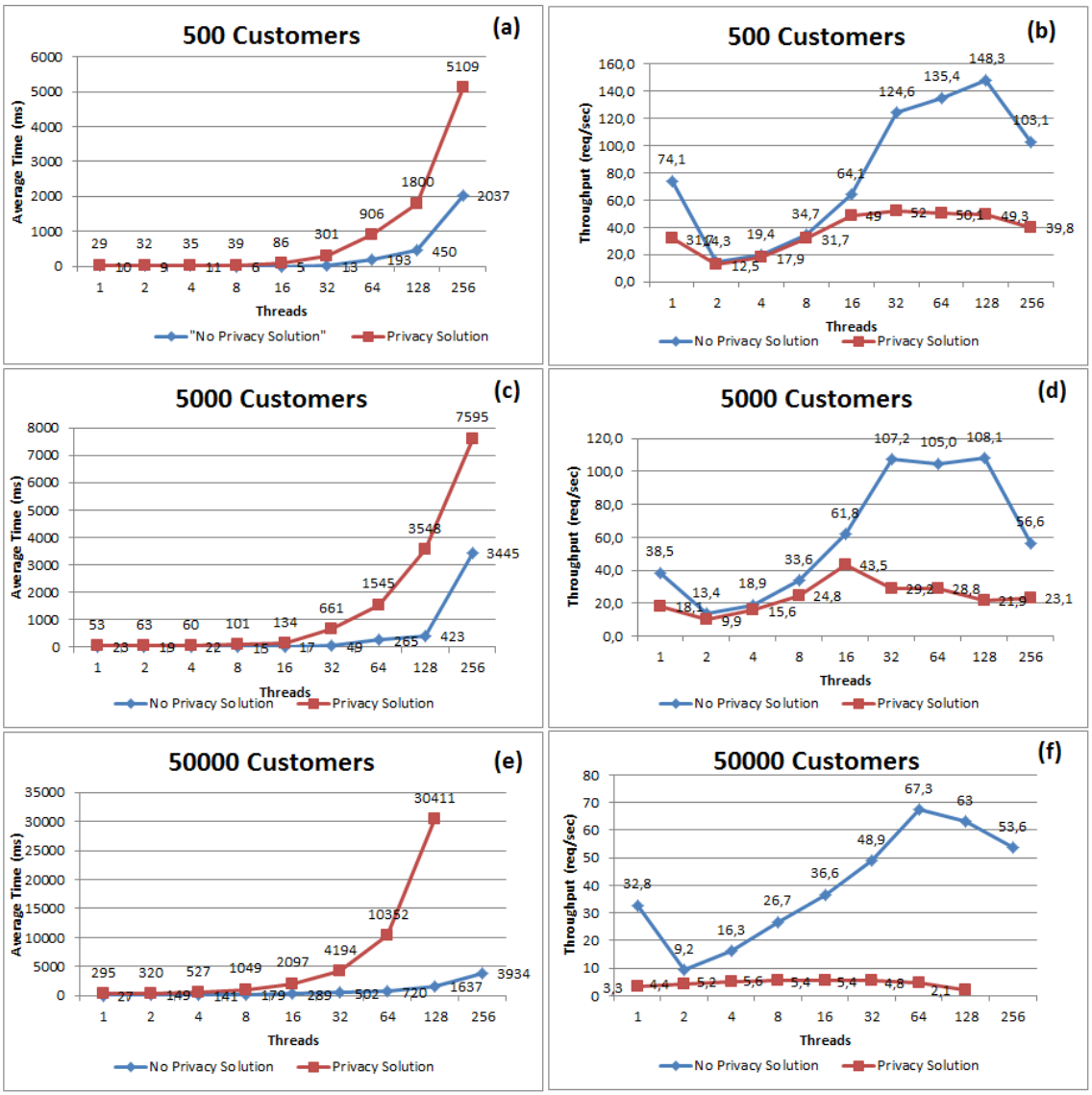


Figure 3. Experiments average time and throughput.

Analyzing the average processing time requests in terms of performance impact, Figures 3a, 3c and 3e shows that the proposed solution has very low impact when few users are using the web application at the same time. Although in some cases the increased time represents a high percentage (for example, in Figure 3a, the average time

for the first sample increases from 10 to 29 ms, representing more than a 100%), the time is milliseconds and this difference is practically irrelevant. So, the average time without the privacy solution is very similar to the other results (applying the privacy mechanism) until around 16 users. As the number of user increases, the differences between both results increase ever more sharply. But, although the inclusion of the privacy solution affects the performance for higher number of users and concurrent access, the system performance is rational for a small to medium enterprises where the number of customers and concurrent access is not too high.

The throughput variation presented in Figures 3b, 3d and 3f is similarly acceptable for the same range: about the 16 first samples the results with and without the proposed privacy solution are similar and the differences arise as it increases. These results reinforce the scalability of the solution for small to medium demands.

Through the analysis of the average processing time requests is possible to observe that, also, the privacy solution is scalable in terms of numbers of records considering the same type of enterprises. Figure 4a consolidates the scalability evaluation experiments, aggregating the average time response results for the respective three amounts of records (50, 5,000 and 50,000 customers) presented in Figures 3a, 3c and 3e, all implementing the privacy protection solution.

In Figure 4a it is possible to observe that the number of records barely affects the system performance for fewer threads. For many threads, the performance is affected in an expected way (the more records, the more processing cost) but the exponential shape brought strong evidence that the mechanism is not feasible to be used in large enterprises and/or high concurrent access situations.

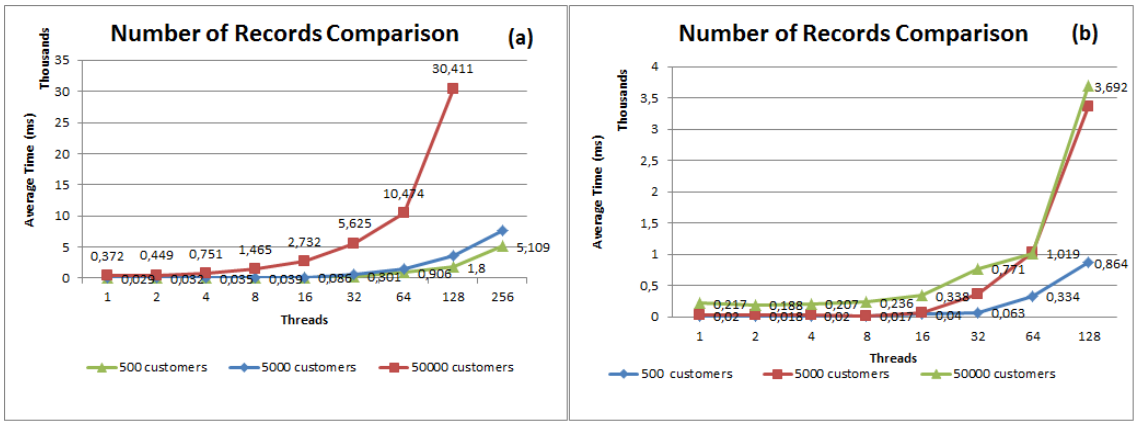


Figure 4. Average time response for different amounts of records implementing privacy solutions.

Figure 4b shows the average time response for the privacy solution of Basso et al (2013). The both solutions have different implementations and respective vantages and disadvantages: in Basso et al. (2013) the authors used AOP (Aspect-Oriented Programming) and the privacy enforcing is done in the application code. The data is recovered from the database and masked in the application. It has the disadvantage of being more vulnerable from the security viewpoint because, face with attacks, the information can be retrieved before masked. Also, it does not consider users preferences for each one of personal information when collecting, storing and using them.

The privacy enforcing mechanism tested in this work is implemented in the database. It is less vulnerable because the data recovered from the database is already masked according to the privacy policies. And it has more resources once it allows users to define privacy preferences for each one of personal information. Although the both solutions were tested in different environments, which influence the results, it is possible to deduce that the current solution produces greater impact on the performance (attention must be taken with the chart scale because, even it is represented on thousands Figure 4a represents 10 times the values in Figure 4b) due to its more elaborated resources.

6. Conclusions and Future Works

Providing efficient and effective privacy policies definition and enforcement for assisting privacy in web applications has long been an open and challenging issue. However, it is critically important for Internet-based data management systems due to privacy protection laws and companies and organizations interests. Introducing solutions for privacy protection in web applications can produce a performance impact, which can determine the feasibility of using the referred solution. So, evaluating this performance impact is essential.

This study presented the definition of a test process model to evaluate the performance impact introduced by privacy protection solutions that were implemented in the database layer. The proposed process is based on performance tests and permits de definition of multiple scenarios, multiple concurrent users (through web virtual users) and different amount of database registers. In comparison with other important process models, our solution has two important features. First, it is generic, i.e., was not developed to a specific type of application and can be used to different web applications segments. Second, it permits performing tests with and without privacy solutions, i.e., permits independent tests and the comparison of the results.

The test process model was validated through experiments involving a web application that simulates an online book store, based on TPC-W benchmark, and a privacy solution that permits users express their preferences about their personal information. The set of tests was useful to allow the identification of under what conditions the use of privacy solution is feasible. A comparison with results of other privacy solution was made and the solution under test was identified as more complete, but with greater performance impact in large scale web systems. This showed that it is more appropriate to be used by companies and organizations where the number of database registers and concurrent access is more limited.

As future work, we intend to complement the tests, evaluating other privacy solutions in the same environment, even to indicate which one is more feasible under the same test conditions. This is one of the areas that we are currently working on.

References

Accenture (2009). “How Global Organizations Approach the Challenge of Protecting Personal Data”. Available: http://www.ponemon.org/local/upload/file/ATC_DPP %20report_FINAL.pdf. Accessed: 19-Mar-2014.

Basso, T., Antunes, N., Moraes, R., Vieira, M. (2013). " An XML-Based Policy Model for Access Control in Web Applications". In proceedings of 24th International Conference on Database and Expert Systems Applications - DEXA, pp. 274-288.

Basso, T., Piardi, L., Moraes, R., Jino, M., Vieira, M. (2014). "A Database Framework for Expressing and Enforcing Personal Privacy Preferences." Paper submitted to The 14th Privacy Enhancing Technologies Symposium – PETS 2014, Amsterdam, Netherlands.

Bertino, E., Lin, D., Jiang, W. (2008) "A Survey of Quantification of Privacy Preserving Data Mining Algorithms". In: Privacy-Preserving Data Mining, vol. 34, C. C. Aggarwal, P. S. Yu, and A. K. Elmagarmid, Orgs. Springer US, pp. 183–205.

Cate, F.H. (2009). "Security, Privacy, and the Role of Law". Security & Privacy, IEEE , vol.7, no.5, pp. 60,63.

Convergência Digital (2012). "Leia a versão do Marco Civil da Internet que foi ao Plenário da Câmara". Portal Convergência Digital, 07 de novembro de 2012. Available: em http://convergenciadigital.uol.com.br/cgi/cgilua.exe/sys/start.htm ?infoid=32316&sid=4#.UZvhwMqNAuv. Accessed: 23-may-2013.

Earp, J. B., Antón, A. I., Member, S., Aiman-smith, L., Stufflebeam, W. H. (2005). "Examining Internet Privacy Policies Within the Context of User Privacy Values", IEEE Trans. Eng. Manag., vol. 52, pp. 227–237.

Exame (2012). "Facebook pode revelar mais sobre dados recolhidos de usuários". Available: http://exame.abril.com.br/tecnologia/facebook/noticias/facebook-pode-revelar-mais-sobre-dados-recolhidos-de-usuarios. Accessed: 24-mar-2014.

Glass, L.; Gresko, R. (2012). "Legislation and Privacy across Borders". International Conference on Privacy, Security, Risk and Trust (PASSAT), 2012 and International Confernece on Social Computing (SocialCom) 2012, pp.807,808.

GloboNews (2012). "Google se envolve em invasão de privacidade na internet". Available: http://g1.globo.com/globo-news/globo-news-em-pauta/videos/t/todos-os-videos/v/google-se-envolve-em-invasao-de-privacidade-na-internet/1818480/. Acessed: 24-mar-2014.

Han, P., Maclaurin, A. (2002). "Do consumers really care about online privacy?", Marketing Manage., vol. 11, no. 1, pp. 35-38.

IETF (2013). "Internet Engineering Task Force (IETF)". Available: http://www.ietf.org/. Accessed: 17-sep-2013.

Jmeter (2014). "Apache JMeter - Apache JMeterTM". [Online]. Available: http://jmeter.apache.org/. [Accessed: 09-jan-2014].

Lemos, R. (2013). "Atrasado, Brasil prepara lei de proteção à privacidade". Folha de São Paulo, 14 de janeiro de 2013. Available: http://www1.folha.uol.com.br/fsp/tec/ 88428-atrasado-brasil-prepara-lei-de-protecao-a-privacidade.shtml. Accessed: 12-mar-2014.

Molinari, L. (2003). "Testes de Software: Produzindo Sistemas Melhores e Confiáveis". Editora Érica Ltda, São Paulo.

Myers, G. J. (1979) “The Art of Software Testing”. John Wiley & Sons, Inc, Canada.

Oracle (2014). “Oracle | Hardware and Software, Engineered to Work Together”. Available: http://www.oracle.com/index.html. Accessed: 24-jan-2014.

Perkins, E.; Markel, M. (2004). "Multinational data-privacy laws: an introduction for IT managers". IEEE Transactions on Professional Communication, vol. 47, no.2, pp. 85,94.

Ponemon (2010). “Economic impact of privacy on online behavioral advertising - Benchmark study of Internet marketers and advertisers”. Available: http://www.ponemon.org/local/upload/file/2010_Economic_impact_of_privacy_on_OBA.pdf. Accessed: 22-may-2013.

Pressman, R. S. (2009) “Software Engineering: A practitioner's approach”. 7th Edition, MacGraw Hill.

Reay, I., Dick S., Miller. J. (2009). “A large-scale empirical study of P3P privacy policies: Stated actions vs. legal obligations”. *ACM Trans. Web* 3, 2, Article 6 (April 2009), 34 pages.

R7 Notícias (2014). “Plenário pode votar Marco Civil da internet nesta semana”. Available: http://noticias.r7.com/brasil/plenario-pode-votar-marco-civil-da-internet-nesta-semana-24032014. Accessed: 24-mar-2014.

Torres, I (2012). “Um ataque contra a sua privacidade”. Revista IstoÉ, Edição 2240, 11 de outubro de 2012. Available: http://www.istoe.com.br/reportagens/245189_UM+ATAQUE+CONTRA+A+SUA+PRIVACIDADE. Accessed: 23-may-2013.

Torres-Zenteno, A.H., Martins, E., Torres, R. S., Cuaresma, J. E. (2006). “Teste de Desempenho em Aplicações SIG Web”. In: Proc. of The Ibero-American Workshop on Requirements Engineering and Software Environments, La Plata, Argentina.

TPC (2014). “TPC-W - Homepage”. Available: http://www.tpc.org/tpcw/. Accessed: 08-jan-2014.

Truste (2013). “Powering Trust In The Data Economy”. Available: http://www.truste.com/. Accessed: 19-Mar-2014.

Veja (2011). “Vida Digital - Congresso dos EUA vai investigar aplicativo para smartphones que 'rouba' dados pessoais”. Revista Veja, 01 de dezembro de 2011. Available:http://veja.abril.com.br/noticia/vida-digital/senador-americano-ira-examinar-escandalo-de-privacidade-envolvendo-smartphones. Accessed: may-2013.

Veja (2013). “EUA: governo vasculha dados de fontes como Google e Facebook”. Available: http://veja.abril.com.br/noticia/internacional/eua-governo-vasculha-dados-de-fontes-como-google-e-facebook. Accessed: 24-mar-2014.

Vieira, M., Madeira, H. (2005). “Towards a security benchmark for database management systems”, in Proceedings, of International Conference on Dependable Systems and Networks, DSN 2005. p p. 592–601.

32º Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos

Florianópolis - SC

XV Workshop de Testes e Tolerância a Falhas (WTF 2014)

Sessão Técnica 2

Algoritmos Distribuídos

Distributed Snapshot algorithm for multi-active object-based applications

Michel J. de Souza[1], **Françoise Baude**[2]

[1]Departamento de Ciência da Computação
Universidade Federal da Bahia (UFBA) – Salvador, BA – Brasil

[2]Univ. Nice Sophia Antipolis, CNRS, I3S, UMR 7271, 06900 Sophia Antipolis, France

michel@dcc.ufba.br, Francoise.Baude@unice.fr

Abstract. *This paper exposes an adaptation of the classic algorithm for consistent snapshot in distributed systems with asynchronous processes due to Chandy&Lamport. A snapshot in this context is described as the consistent set of states of all involved communicating processes that allows recovering the whole system after a crash. The reconstructed system state is consistent, even if messages injected into the system from the outside while the snapshot was ongoing may have been lost (if such messages can not be replayed). We expose how to adapt this algorithm to a particular distributed programming model, the Active Object model (in its multi active version). We applied it successfully to a non trivial distributed application programmed using Active Objects serving as a publish/subscribe and storage of events middleware, dubbed the EventCloud.*

1. Introduction

When a system is dealing with a huge amount of data, it is important to have a backup mechanism that allows its recovery in case of disaster, loss of energy, etc, which in the sequel, we assume requires recovering the whole system, not only one or a few processes. In practice, to accomplish this task a snapshot algorithm can be used to regularily take a consistent and global, even if distributed, checkpoint. A snapshot is indeed the process of collection of data that allows to create a global consistent image of the system state. This paper proposes a snapshot algorithm that works as an adapted version of the classic Chandy-Lamport algorithm [Chandy and Lamport 1985]. The algorithm is adapted in order to suit the constraints from the use of the Active Object programming model that is used to program our distributed system dubbed the EventCloud. Its aim is to act as a storage of events, and allow end-users to retrieve them in an asynchronous manner thanks to a publish-subscribe model. The system is architectured along a structured peer-to-peer overlay network, and aims to be deployed on distributed physical infrastructures (private cluster, private/public cloud, etc.).

Snapshot algorithms assume initial input data is available and could be used ultimately to recover the execution from the very beginning if needed. On the contrary, in our case, input data is coming continuously from the outside, in the form of publications and subscriptions injected inside the EventCloud. It is not realistic to assume we can log all this exterrnal information (as the log would replicate the data that is eventually stored in the EventCloud). Consequently our solution can suffer of publications loss in some circumstances; however, we propose an ad-hoc solution for preventing subscriptions loss.

This paper is organized as follows: the section 2 provides the background and some important concepts to understand the work proposed; the section 3 presents the solution in details and how to address the specificities due to the Active Object programming model as used to implement the EventCloud; the section 4 presents the way an EventCloud gets reconstructed out of a taken snapshot, and consequently which are the failure supported assumptions. The last section concludes and summarizes the improvements left as future works.

2. Background

The proposed snapshot algorithm is implemented within the EventCloud software using the ProActive middleware , thus offering both large-scale deployment capabilities of Active Objects on distributed infrastructures and remote communication features between these objects. This section describes the needed concepts.

2.1. Proactive

ProActive [INRIA and UNS 2000] is an open source Java library aiming to simplify the programming of multithreaded, parallel and distributed applications for clusters, grids, clouds, and any distributed infrastructure in general.

Some ProActive features useful in the sequel are the following. Activities that can execute in parallel take the form of distributed, remotely accessible objects. An active object is an entity with its own configurable activity. Each active object has its own thread of control (so the term "active") that is in charge of selecting and serving the next method call, among those awaiting to be served. Requests to execute method calls are automatically stored in a queue of pending requests. Requests are sent in a FIFO manner on the communication channel connecting a caller and a callee. This FIFO behaviour happens because of a Rendez-Vous protocol: an acknowledgement from the callee is sent back to the caller, to commit the request is stored in the request queue waiting to be served in an asynchronous manner. Consequently, the caller can not initiate a second request before receiving this acknowledgment, ensuring FIFO placement of corresponding requests in the queue in case they target the same callee. By default, the serving policy of requests stored in a queue is FIFO, and one request is served by the control thread at a time. Eventually, active objects interact by asynchronous method calls. Results of method calls are called futures and are first class entities. Callers can transparently wait for results using a mechanism called wait-by-necessity [Caromel 1993]. A callee transparently updates the corresponding future of a method invocation, which automatically wakes up the awaiting caller if it is already blocked waiting for the result.

A distributed or concurrent application that uses ProActive is composed of several active objects. A recent ProActive version [Henrio et al. 2013] replaces active objects by multi-active objects, allowing to serve more than one request at a time. To accomplish this task, one can declare compatibility groups of methods. Requests marked as compatible are allowed to be served in parallel. The algorithm proposed in this work is implemented using the multi-active objects version.

2.2. EventCloud

As described in [Baude et al. 2011], the EventCloud is a distributed store of data following the W3C RDF specification, that handles SPARQL queries across the distributed

architecture. RDF is a fundamental part of the Semantic Web stack offering a graph structured data model where each relation is made by default of a triple (subject, predicate, object). Although RDF uses triples, the EventCloud works with quadruples - an extended version of a triple that adds a fourth element (usually named context or graph value) to indicate or to identify the data source. Events stored in the EventCloud module can be simple or compound. A compound event is made of a set of quadruples with each quadruple sharing the same graph value. This graph value is assumed to be an identifier that uniquely identifies a compound event.

The EventCloud is a module deployed over a CAN [Ratnasamy et al. 2001] structured peer-to-peer network (a D-dimensional space torus topology). Each peer is a multi-active object that can receive requests that are put into a request queue. All the non compatible requests (e.g. adding a peer reference as neighbour request, and routing a request towards are served in a FIFO (First-in, First-out) manner, whereas declared compatible ones can be served in parallel by the multi-active object (e.g. several requests to route a message whose purpose is relative to a publish or subscribe operation in a dataspace zone the peer is not responsible of can be handled in parallel: their treatment consists to forward the message further in the CAN, by sending it to the right neighbour). Each peer has also a handful of Jena datastores [Jena 2014] to store quadruples, intermediate query results, queries, etc. In the sequel, we will simplify by considering only one single datastore is attached to each peer.

The EventCloud works with the publisher-subscriber paradigm where any external user connected to the EventCloud through a proxy can subscribe to receive data that he is interested in, or can publish data. To make this possible, the EventCloud uses the SPARQL query language to formulate the subscriptions.

The EventCloud module integrates within the PLAY Platform described in [Roland Stühmer et al. 2011]. The PLAY Platform aims at creating an Event Marketplace at the Web scale. A marketplace works as a search engine dedicated to events and provides visibility to distributed sources of events.

2.3. Chandy-Lamport snapshot algorithm

Before starting to explain how the snapshot algorithm of [Chandy and Lamport 1985] works, it is important to establish some concepts used by it:

- A distributed system consists of a finite set of processes with finite set of channels. It is described by a directed, labeled graph in which the vertices represent processes and the edges represent channels.
- A channel is the connection between two processes by which messages can be sent. It is assumed to have infinite buffers, to be error-free and to deliver messages in the order sent.
- The state of a channel is the sequence of messages sent along the channel, excluding the messages received along the channel.
- A process is defined by a set of states: an initial one and a set of actions. An action a in a process p is an atomic action that may change p's state itself and the state from at most one channel c incident on p.
- The communication in a distributed system happens by the exchange of messages. A process only records its own state and the messages it sends and receives.

- To determine a global state, a process must count on the cooperation of the others processes composing the system. They must record their own local states and send this data to a chosen process charged of recording the collected data.
- Unless relying upon a global common clock, the processes cannot record their local states precisely at the same instant.
- The global-state-taking algorithm (snapshot) must run concurrently with the underlying computation not altering it.
- A global state for a distributed system is a set of process and channel states. The initial global state is one in which the state of each process is its initial state and the state of each channel is the empty sequence.

Marker-Sending Rule for a Process p. For each channel *c*, incident on, and directed away from *p*:

p sends one marker along *c* after *p* records its state and before *p* sends further messages along *c*.

Marker-Receiving Rule for a Process q. On receiving a marker along a channel *c*:

if *q* has not recorded its state **then**
  begin *q* records its state;
    q records the state *c* as the empty sequence
  end
else *q* records the state of *c* as the sequence of messages received along *c* after *q*'s state was recorded and before *q* received the marker along *c*.

Figure 1. Key rules of the Chandy-Lamport snapshot algorithm [Chandy and Lamport 1985]

The algorithm uses two basic procedures to control the global state construction process and can be initiated for one or more processes. The process in charge of initiating the snapshot algorithm taking records its state spontaneously without receiving a token from others processes. The procedure that represents the sending of a token is named *Marker-Sending Rule*. In this procedure, a process will initially record its state and then will send a token along each channel c directed away from itself. The procedure that represents the receiving of a token is named *Marker-Receiving Rule*. On receiving a token for the first time, a process will record its state (locally). Once a process receives the remaining tokens (from its neighbors), it needs to record the channel state by which the token has arrived. This state is composed by the messages that has arrived after the process has recorded its state and before the receiving of the token. Once the data is collected each process sends it to the process that has started the algorithm. The Figure 1 depicts the algorithm.

3. Snapshot algorithm for the EventCloud system based on Active Objects

3.1. Snapshot Manager

The Snapshot Manager regularly triggers the execution of the snapshot algorithm. If asked, it can initiate an EventCloud instance reconstruction by using the data collected, so it is working as a safe storage place receiving all the peer's data, etc. Reconstruction will be addressed in the next section.

The Snapshot Manager is a multi active object having a reference to an Event-Cloud Registry. The EventCloud Registry holds reference to EventCloud instances. In this way, whenever the Snapshot Manager needs to perform any operation (start a snapshot, restoration, etc) for one specific Event Cloud, it is going to contact the registry in order to obtain the reference that will allow to fulfill the task.

3.2. Snapshot algorithm adapted from Chandy-Lamport solution

As within Chandy-Lamport algorithm, in the adapted solution presented in this work, channels are assumed to be uni-directional, but between any two peers, a channel in both directions does exist. Moreover, there is a channel from a peer to itself, as a peer can send requests to itself. Differently from original version, each peer collects its data and sends it to the safe storage place; whereas in Chandy-lamport algorithm, the data collected is sent to the peer that has started the algorithm.

Peer data to be saved with the algorithm are as follows:

1. Jena datastore - application data.
2. Request queue relevant information.
3. Topology information: neighborhood table, zone, etc.
4. Some other EventCloud management parameters.

The Figure 2 above depicts our adapted snapshot taking algorithm.

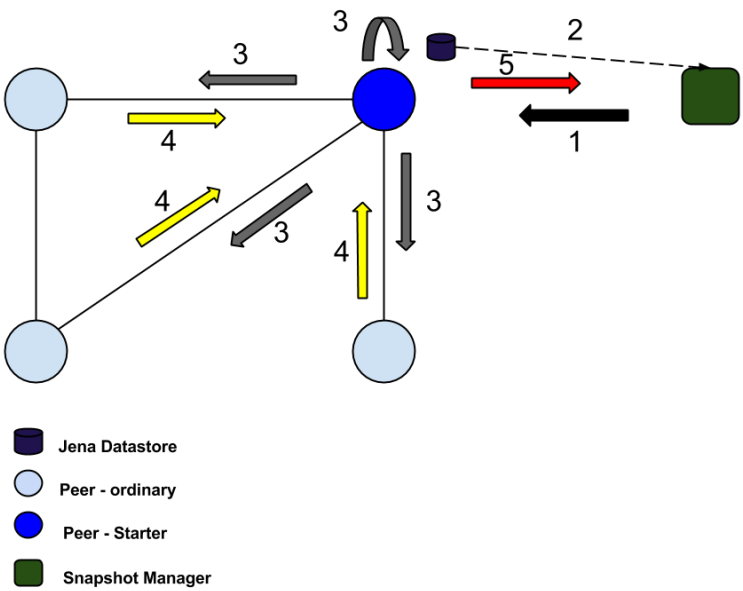


Figure 2. Snapshot algorithm execution process. Depicted from only the starter peer viewpoint.

The **black arrow** (1) illustrates the moment when the Snapshot Manager initiates the snapshot process choosing randomly a peer to trigger the algorithm.

Once the chosen peer to trigger the algorithm receives the request, it records part

of its state by sending its Jena datastore[1] to the safe storage place (Snapshot Manager) - this step is illustrated by the **dashed black arrow** (2). At this point, the peer locally copies part of the data it needs to collect - topology information (neighborhood table, zone, etc), in order to add this data to its corresponding saved state later on (see below).

Once this same peer has performed the tasks described above, it propagates the token towards its neighbors (including to itself). This step is described by the **gray arrows** (3). This ends up the operation named *Marker sending rule* of the Chandy Lamport algorithm (see Figure 1).

On the picture, a **yellow arrow** (4) describes a sending of a token from all the starter peer neighbors to it. This represents only the starter peer perspective. However, on receiving a token for the fist time, a peer will send a token to each one of its neighbors. Notice the figure only depicts the initiator peer algorithm execution, but it runs almost simarily on each peer (except regarding the black arrow).

Regarding our adaptation of the *Marker receiving rule* of the algorithm, the following is taken in consideration. The **grey** (3) or **yellow arrows** (4), the receiving of a token, trigger a key action: ”closing the channel” they are sent onto.

Every peer, including the initiator one, will have to have served n+1 tokens in total (token carried in the form of a remote method invocation), to terminate the execution of the algorithm instance. One token from each of its n neighbors, and another one that it has sent to itself (as each peer might trigger a method call, i.e. send a request, to itself). Indeed, in the Active Object model, the active object can invoke methods on itself; in the EventCloud implementation, this is extensively used. This self-sent token closes a channel from it to itself. In general, each token served by a peer closes the channel directed from the corresponding sender peer to it. Notice that because each peer is an active object, it stores the received requests (messages) in its request queue. Consequently, we have to adapt the Chandy Lamport algorithm to the fact that channels are not easily identified: they are all mixed in each active object single request queue, and are materialized only at the receiving edge of the channel (i.e. if peer A can send a method call MC to peer B because A knows B’ object reference, the unidirectional communication channel that connects A to B gets materialized by the fact that the message MC is stored in the request queue of peer B. Before MC gets served, one can consider that MC is still on the wire between A and B. This is why MC might need to be part of the global snapshot, besides the states recorded at the moment A and B respectively and independently serve their first token).Consequently, the operation to save in transit messages then close a channel once a token from a neighbour has arrived, has been carefully designed, as detailed just below.

One really important information of the global snapshot to be saved is the relevant (subpart of) request queue information of each peer; so, in background, always before the scheduler in charge of selecting which next request(s) can be served takes out a request from the peer queue, an evaluation process happens to decide if this request (the corresponding message) will be locally copied with all the other requests that represents the in-transit messages (these requests will be sent to the Snapshot Manager once the peer has

[1]Jsch [jsc 2014] is a Java implementation of SSH2. It allows to connect to a server and uses port forwarding , X11 forwarding, file transfer, etc. As Jsch doesn’t allow folder transmission, to be able to transfer the Jena datastore it’s necessary to zip its folders.

finished its snapshot taking process). For a request to be saved, three properties need to be verified: the request cannot be a token because a token represents only a control request part of the algorithm managing what needs to be saved; the peer that has received the request needs to be in RUNNING state what implies the peer being active from the snapshot taking algorithm viewpoint and indicates it hasn't yet finished it's data collection of the current snapshot algorithm instance; at last, the directed channel from the peer sender to peer receiver needs to be still open. The reception of a token on a channel closes this channel. Hence, if a peer has sent a token to another peer and after generates new requests to this same peer they won't be saved at receiver side because they have arrived after the token that closed the channel (a strong property of the active object model is that communication between any two objects is FIFO). On the contrary, if a peer has sent a token to a neighbour, but has not received a token from it, any received request originating from this neighbour is to be considered in transit. So it is going to be part of the corresponding channel state in the global snapshot.

At last, having received all the needed tokens, the peer will send the data it locally collected to the Snapshot Manager, thus enriching the peer state that was only partly recorded so far. This data is composed by the relevant request queue information saved during the moment at which the peer gets engaged in the algorithm snapshot (it has received the first token) and the moment it finishes (it has received all the tokens) and the topology information copied previously. This step is described by the **red arrow** (5).

The picture above depicts the snapshot algorithm from only one peer perspective. However, after the other peers have received the token, they will execute the same set of actions (symmetrically): save its state, send a token to the neighbors, etc. The main difference lies in the snapshot algorithm initiation: the Snapshot Manager contacts only the starter peer (the one that triggers the whole algorithm process). After receiving the token each starter peer neighbor will initiate the algorithm. The algorithm ends when each peer has received all the tokens from its neighbors and has sent the data collected (the peer state, neighborhood table, etc) to the Snapshot Manager. The Snapshot Manager is able to detect when a snapshot taking is terminated, because, for each EventCloud instance, the number of peers part of the distributed support is known (it is stored as information in the EventCloud registry). Until it has not received the expected number of peer states, this snapshot instance is not valid to be used for a recovery.

3.2.1. Intuition about the corrrectness of the algorithm

To gain an intuitive understanding of some points into the algorithm, let us devise a scenario with an Event Cloud having 3 peers that exchanged requests that may not all be served when the snapshot taking starts. Two of them are faster than the remaining one, that is, serve their requests in a faster manner: they have already exchanged tokens between themselves while the third peer hasn't treated yet its first token. Each of these two faster peers has already saved its state locally (Jena datastore, zone, neighborhood table, etc) during the first token service; now each proceeds serving requests which may generate new requests sent to the slower peer that hasn't already saved its state. In the meantime, received requests sent from the slower peer gets ready to be served by the faster peers, and they will be tagged as in-transit.

More globally, the question we raise is dictated by the fact that in our setting, snapshot taking happens concurrently with the application execution: in a snapshot (set of local peer states or channel states), can it happen that it contains a given information that is not a consequence of the recorded global state ? If yes, this means the system could be recovered from a snapshot that is not consistent, in the sense that it reflects a situation that the system would never be able to reach if restarted from an older snapshot. It is well-known that in a distributed system, message reception suffers from non-determinism of arrival, this explains why different executions and so, different peer states could arise from a same global snapshot restart. Also, these different executions may not exhibiting the same requests, nor the same request contents. Because our snapshot comprises copy of requests tagged in-transit, we must ensure that it is correct to serve them in the recovered execution. Also, because the snapshot comprises peer local state copies, we must ensure that in these states, only information that is the result of what the snapshot encompasses is present: otherwise, this means the state encompasses causal effect of actions but these actions are not supposed to have arisen as summarized by the snapshot content.

In the original Chandy-Lamport algorithm, there is no need to raise this question: a process engaged in the algorithm is applying rules depicted in Figure 1, but in the meantime, it is not processing any of the messages received on its incoming channels, meaning the state that is distributively recorded in the snapshot is not evolving while it's built. On the contrary, in the Active Object model, a peer must proceed serving the requests (in FIFO order by default) to have a chance to process tokens. That means the numerous tokens related actions like local state and channels state savings can happen while peer state has been affected by applicative-level requests service that must eventually not be "visible" (directly or indirectly) in the snapshot. So, for the algorithm to be correct, it must ensure that these effects are left aside and are not wrongly recorded in the snapshot.

First, peer local state saving happens at the very first token service. It only reflects request service effects that must be part of the snapshot. In particular, none of the served requests yielding to the recorded state was tagged as an in-transit request. Indeed by definition, in-transit requests stored in the queue are positioned after the very first token in the request queue, so always served after the peer local state saving, thus having no impact on the saved local state. Also, no request served before the first token service could correspond to a request that should not be part of the global state because was created by a sender after it recorded its own local state (as communication channels in ProActive are FIFO).

The second case discussion is about channel states that are logged, that correspond to in-transit requests within the snapshot. Could it happen that an in-transit request gets recorded in the snapshot alas it should not have been because its originating action is not part itself of the snapshot ? The risk being that in a replay of the system, the execution takes place in a way such that the request that triggered the one tagged in transit and part of the recovered state does not arise. The answer is no: No in-transit request can be a consequence of a request service whose effect is not part of the snapshot.

Thirdly, while the snapshot taking is running on the distributed system, active objects do not stop serving requests, some being detected in-transit requests, some being received before (resp. after) locally entering the snapshot taking execution (so before (resp. after) the first token service). Except for those received before first token service,

no impact from their service happens on the snapshot recorded state: local state was copied while serving the first token, which atomically propagated tokens on outgoing channels that, once served will close channels. Consequently, any new request sent on these channels (assumed to be FIFO) cannot affect the receiver recorded local state nor corresponding channel state.

3.3. Current limitations

If peers are removed or added to an EventCloud intentionally, according to its associated load balancing strategy, removal or addition of peers is an information that can easily be tracked for recording the effective number of peers. However, notice that supporting removal or addition of peers while a snapshot algorithm is running is left as future work (see section 5). Say another way, the algorithm as it is presented here does assume the number of peers is static, in the sense the topology of the CAN architecture is not changing while the snapshot algorithm is running.

In the current implementation, only one instance of the snapshot algorithm can be running at a time. To initiate the execution of new snapshot algorithm instance the termination of its execution must have been detected. This is a limitation because we may want to periodically put in execution a new algorithm instance, that is, after a certain amount of time a new instance would automatically start, getting a certain overlapping of many global different states. As recovery is triggered from the most recent completed snapshot, it is obvious the less time elapsed since its completion, the best it is as only ”few” work is going to be lost and redone. In fact, to support many instances of the snapshot algorithm run in parallel would simply require to extend the current implementation by making all exchanged tokens and associated treatments triggered by the snapshot algorithm tagged by an instance number. For instance, a channel may be closed by a token service of one specific instance, while it may still be open regarding another instance of the snapshot algorithm execution.

4. EventCloud recovery

Given the snapshot algorithm has allowed to record all the data needed, our next aim is to support the inverse process, that is, reconstruct the system using this information.

4.1. Restoration process

When one decides to reconstruct an EventCloud, the Snapshot Manager will trigger the restoration process. According to the snapshot identification, the Snapshot Manager will access the file that contains management information about the EventCloud in question and the state set (from each peer) that contains all the data: neighborhood table, peer request queue, Jena datastore, etc.

The reconstruction algorithm starts similarly to the process of an EventCloud instantiation: when an EventCloud is instantiated, the peers are created with random identifiers, they do the join process according to the CAN protocol to establish the zone they are responsible for and the neighborhood table, etc. However, in the reconstruction process, the peers will be created with well known identifiers, passed as arguments, as this information is already established. Another difference lies in the join phase that is discarded. This phase is useless when reconstructing an EventCloud because the topology information (neighborhood, zone, zone split history) from each peer is already known.

Once the peers created, the data collected must be sent to the peers and some adjustments must be also done. First, the Snapshot Manager sends the corresponding Jena datastore, by using Jsch library, to each peer. This process will create the needed local connection with Jena datastore. Then, the topology data is sent. The topology data is composed by the neighborhood table, peer zone and the split history; however the active object's remote addresses (handled within a stub object) entries in the neighborhood table from each peer is not valid because all the peers were recreated. So, the stub entries need to be updated. To fulfill this task, each peer sends a 2-tuple (id,stub) to its neighbors. Receiving this 2-tuple, the peer will replace the stub in the neighborhood table that corresponds to the id, concluding the update. The list of 2-tuples (id, stub) was created in the peer creation phase. As described before in this section, the join phase in the reconstruction process is useless because the zone and split history are passed to the peers (they were collected). When the peer has received all these topology information it needs to be activated. Consequently, the Snapshot Manager will send a request to turn the peer activated which has as an important side-effect to populate the front of its request queue[2] with the in-transit recorded requests. There is no specific per-channel information, as in-transit requests were recorded in the order they were identified, whatever incoming channel they originated from. Hence, the EventCloud is reconstructed, active and ready to receive even additional new requests and serve all of them.

4.2. Characteristics of the supported failure model

4.2.1. Risk of information loss

Let's suppose that the snapshot algorithm instance finishes and suddenly one, some or even all peers in the system crash. From the moment of the crash and the system recovery from the last saved global snapshot, the risk is to miss some of the published events, or some subscriptions. While missing events might be catastrophic if they correspond to critical alerts (e.g. from a nuclear plant), in some other situations such loss may be affordable (e.g. if events are used to signal the arrival of new information to subscribed users of a social network).

Recovering published events is impossible if they were injected in the EventCloud at the following moment: before a crash and no peer was able to serve the corresponding request(s) before it terminated its snapshot taking process. Say differently, there is no effect of the published event on the snapshot constituted by the peer and channel saved states. In practice, it is not realistic to log all events injected into an EventCloud, as it might mean logging the entire actions that happen outside the EventCloud, in the external world possibly at web scale.

Subscriptions might be less numerous than publications. Moreover, when a user submits a subscription, the aim is to be kept informed of matching publications that can happen in the future. A subscription stands as a permanent request until it gets removed

[2]As peer activation is not an atomic process, it could happen that a peer is activated, serves an in-transit request that generates a fresh request that stands in the neighbour queue at the first place, before the logged in-transit ones. It is thus important to force the logged in-transit ones to be put at first, then only activate the peer allowing it to start serving requests: first the logged ones, then the fresh ones. Letting fresh ones bypass recovered in-transit ones would break the FIFO nature of ProActive channels.

by the subscriber. It is thus important to devise a practically feasible[3] solution to prevent any loss of them. The proposition is as follows.

It consists in a subscription-specific automatic retry mechanism, based upon an extension of the existing EventCloud proxies [Roland Stühmer et al. 2011]. In more details, it could leverage the fact that the EventCloud relies upon the assumption that all peers participating in the network run a synchronized clock (this is a reasonable assumption for an EventCloud deployed on an administered infrastructure such as a cluster within a data center). As such, it is possible to tag the last snapshot with the most recent clock value corresponding to the moment all peer states have been fully stored. The proxy is in charge of contacting a peer (known as a tracker in P2P terminology) of the EventCloud, to deliver to it the subscription. Each subscription (and event) entering the EventCloud is gaining the clock value at the moment it enters the EventCloud (because the publish-subscribe matching algorithm requires that information) through the contacted peer. Given that, if an injected subscription gets a clock that is greater than the last checkpoint saved clock value, there is a risk in case of failure, that it gets definitively lost. At the proxy side, knowing the clock value associated to the subscription is easy: the contacted peer sends back an acknowledgment of receipt to the proxy (in the form of either a reply updating the future associated to the initiated method call at proxy side, or as a new request from the peer towards the proxy, as a proxy is also an active object). Obviously if the contacter peer fails before the acknowledgement gets back to the proxy, the proxy could after a timeout automatically retry the delivery, once it detects the EventCloud recovery is over. In all cases, even if not facing a crash of the EventCloud, the proxy could keep a copy of the subscription until being sure there exists a more recent snapshot that involves it. Contacting the snapshot manager allows the proxy to know about the latest saved snapshot clock. Before being sure of that, proxy could regularily reinject the subscription in the EventCloud, not mandatorily contacting the same peer as peer clocks are synchronized. Given each subscription is uniquely identified, any peer receiving it (because it holds a zone that should index events possibly matching the subscription) could thus avoid duplicating it in the corresponding Jena datastore thanks to this unique identifier.

4.2.2. Discussion about partial or whole EventCloud recovery

In search for a better availability of the EventCloud offered as a service to its end-users, one would like to get a fault-tolerant version where only failed peers and not the whole EventCloud is restarted in case of peer failure.

This might be feasible by using a ProActive version implementing the general purpose checkpoint and recovery protocol devised in [Baude et al. 2007]. Indeed, [Baude et al. 2007] proposes a protocol by which a new instance of a snapshot algorithm is triggered periodically. This protocol ensures that no message internally exchanged between active objects is lost even in case of some failure takes place. The main challenge solved by [Baude et al. 2007] is how to correctly recover – only– the failed active objects' state of the system given resulting global checkpoint might indeed correspond to an inconsistent global state. To cope with this challenge and restore the system consistently each

[3]i.e. that avoids systematically logging all information (including publications) that are injected from the external world into the EventCloud

request exchanged in the execution needs to be tagged by the number of corresponding ongoing checkpoint it should belong to. So the need to rely upon a specific "fault-tolerant" version of the ProActive library which today is not usable with multi-active objects. On the contrary, the Chandy-Lamport algorithm adaptation we propose here doesn't assume the reliance upon a modified Active Object programming library: the support of the snapshot messages is simply programmed by extending the by-default request serving policy of active objects. The mentioned request tag is useful in the checkpointing protocol to turn the necessary requests (those identified as violating the global consistency of the global checkpoint) into promised requests: a recovered peer blocks if trying to serve a promised request, until this request gets filled by the data extracted from its replay. These automatic blocking operations and more tricky actions (see details in the paper) eventually ensure the recovered global state of the system becomes globally consistent.

Notice that this fault-tolerant solution suffers from the same limitation pertaining to possible loss of external information sent to the fault-tolerant application (i.e. the EventCloud), as we discussed in 4.1. In fact, if an active object fails before serving a request which could not have the chance to be part of any snapshot, there is no practically feasible solution to recover it even restarting the execution of the application from the very first state, i.e. from the initial state.

Differently from the aforementioned solution, the solution proposed here constructs a succession of global states which are consistent, and from which a direct recovery of the whole system gets feasible.

5. Conclusion and perspectives

In this paper, we have presented an adaptation of Chandy-Lamport Snapshot algorithm that works as a best-effort fault-tolerant solution. To validate the work, the reconstruction of an Event Cloud was made by using the data collected from the snapshot recording and verifying if the reconstructed version had the same features of the initial one (same neighborhood set, Jena data, requests queue, etc).

However some improvements can be done. From a pure performance viewpoint, the snapshot algorithm uses a Java-based library to transfer all the data collected to the safe storage place which could be lowered by using a more native approach.

At this point, our implementation doesn't allow the initiation of multiple instances of snapshot being taken from the same Event Cloud at the same time, that is, to start a new algorithm instance the first one needs to be completely finished. More fundamentally, current implementation assumes that the EventCloud architecture is static that is, peers are not assumed to leave or join the P2P network CAN dynamically. Supporting dynamicity requires not only to modify the implementation, but first to devise a slight variant of the protocol itself. Indeed, a channel in the snapshot taking process is closed once a token from corresponding sender gets served. Allowing the sender to intentionnally leave the network while our distributed algorithm runs may imply no token gets sent from its side. An extension of our proposition could be to enforce a gently leave, i.e. informing the neighbours about the intention to leave before effectively leaving the peer-to-peer network. Symmetrically, allowing a peer to join the CAN network means neighbouring sets of neighbour peers grows while a peer may be already participating in a snapshot taking execution. How to consistently incorporate this new peer state in the global snapshot is

also to be first devised at the protocol level.

Acknowledgements This work received the financial support of the PROGRAMA CIÊNCIA SEM FRONTEIRAS, and the European STREP FP7 project PLAY www.play-project.eu.

References

(2014). Jsch library. `http://www.jcraft.com/jsch/`.

Baude, F., Bongiovanni, F., Pellegrino, L., and Quéma, V. (2011). PLAY Deliverable D2.1 Requirements specification for Event Cloud Component. Technical report, `http://www.play-project.eu/documents/finish/3-deliverables-final/20-play-d2-1-requirements-event-cloud/0`.

Baude, F., Caromel, D., Delbé, C., and Henrio, L. (2007). Promised messages: Recovering from inconsistent global states. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 154–155, New York, NY, USA. ACM.

Caromel, D. (1993). Toward a method of object-oriented concurrent programming. *Commun. ACM*, 36(9):90–102.

Chandy, K. M. and Lamport, L. (1985). Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75.

Henrio, L., Huet, F., and István, Z. (2013). Multi-threaded active objects. In Nicola, R. and Julien, C., editors, *Coordination Models and Languages*, volume 7890 of *Lecture Notes in Computer Science*, pages 90–104. Springer Berlin Heidelberg.

INRIA and UNS (2000). Proactive parallel programming suite. `proactive.inria.fr`.

Jena (2014). Apache jena. `http://jena.apache.org`. Last access made in 1st March 2014.

Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S. (2001). A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev.*, 31(4):161–172.

Roland Stühmer et al. (2011). PLAY Deliverable D1.4 Play Conceptual Architecture. Technical report, `http://play-project.eu/documents/finish/3-deliverables-final/19-play-d1-4-conceptual-architecture/0`.

Uma Solução Autonômica para a Criação de Quóruns Majoritários Baseada no VCube

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

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

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

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

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

Abstract. *This paper presents an autonomic solution for building and maintaining quorums in distributed systems. The processes are organized in a virtual hypercube topology called VCube. The topology is constructed dynamically based on the information obtained from a system monitoring. Faulty process are removed from the system and a new configuration is created using only the correct processes. Processes can fail by crash and a failure is permanent. The quorum size and load generated are well distributed and the reconfiguration is automatic, tolerating up to* $n-1$ *faulty processes. The proposed system was compared with a solution using a binary tree. Experimental results confirm the resilience and stability of quorums in the VCube.*

Resumo. *Este trabalho apresenta uma solução autonômica para a construção e manutenção de quóruns em sistemas distribuídos. Os processos são organizados em uma topologia virtual de hipercubo chamada VCube. A topologia é construída dinamicamente com base nas informações de falhas obtidas de um sistema de monitoramento. Processos falhos são eliminados do sistema e uma nova configuração é criada utilizando apenas os processos corretos. Os processos podem falhar por crash e uma falha é permanente. Os quóruns gerados possuem tamanho e carga bem distribuídos e a reconfiguração é automática, tolerando até* $n-1$ *processos falhos. O sistema proposto foi comparado com uma solução em árvore binária. Resultados experimentais confirmam a resiliência e estabilidade dos quóruns no VCube.*

1. Introdução

Sistemas de quóruns foram introduzidos por Thomas (1979) como uma solução para coordenar ações e garantir consistência em um sistema de banco de dados replicado. Considere um sistema distribuído como um conjunto finito P de $n > 1$ processos independentes $\{p_0, ..., p_{n-1}\}$ que se comunicam usando troca de mensagens. Um *sistema de quóruns* em P é um conjunto de subconjuntos de P, chamados *quóruns*, no qual cada par de subconjuntos tem uma intersecção não-vazia [Merideth e Reiter 2010]. Garcia-Molina e Barbara

(1985) estenderam esta definição introduzindo o conceito de *coteries*, isto é, grupos exclusivos. Este modelo inclui a propriedade de minimalidade, na qual nenhum conjunto contém outro conjunto do sistema, o que o torna mais eficiente.

O algoritmo proposto por Thomas implementa um mecanismo tradicional de votação no qual o vencedor é o valor com a maioria dos votos. Cada atualização em um item de dados é marcada com um *timestamp* e executada em um quórum composto pela maioria dos servidores. Quando um cliente precisa ler um valor do banco, ele acessa novamente um quórum com a maioria das réplicas e escolhe o item com o *timestamp* mais atual. Em um sistema com três servidores, por exemplo, se a escrita for replicada em quaisquer dois servidores, toda leitura posterior em no mínimo dois servidores retornará o valor mais atual, isto é, a intersecção dos quóruns garante que, no mínimo, uma replica terá o item de dado mais recente. O trabalho de Thomas considera que cada processo contribui de forma igualitária na votação. Gifford (1979) estendeu este modelo atribuindo pesos aos processos. Além disso, os quóruns são divididos em duas classes, leitura e escrita, e somente quóruns de classes diferentes precisam se intersectar.

Além da replicação de dados [Liu et al. 2011, Abawajy e Mat Deris 2013], sistemas de quórum têm sido utilizados por diversas outras aplicações, como exclusão mútua [Maekawa 1985, Fujita 1998, Atreya et al. 2007, Naimi e Thiare 2013], comunicação em grupo [Tanaka et al. 1999], controle de acesso seguro [Naor e Wool 1996], e sistemas tolerantes a falhas bizantinas [Liskov e Rodrigues 2006, Merideth e Reiter 2007, Merideth e Reiter 2008]. Por exemplo, em uma solução de exclusão mútua baseada em pedidos de permissão sem a utilização de quóruns, se um processo deseja obter acesso a um recurso compartilhado, ele deve enviar uma mensagem de requisição para *todos* os outros processos do sistema e aguardar pelas mensagens de permissão [Ricart e Agrawala 1981]. Neste caso, um sistema de quórum pode reduzir o total de mensagens, visto que este mesmo processo precisa enviar a requisição apenas para um subconjunto dos processos que forma um quórum. Os processos que respondem ao pedido de permissão ficam bloqueados para novos pedidos até que o recurso seja liberado. A intersecção entre os quóruns garante a integridade da exclusão mútua.

Em geral, as soluções de quóruns são avaliadas em termos de tamanho, carga e disponibilidade [Vukolić 2010]. O tamanho é o número de processos em cada quórum e a carga indica em quantos quóruns cada processo está contido. A disponibilidade está ligada ao impacto das falhas no conjunto de quóruns do sistema. O modelo de Maekawa (1985), por exemplo, gera quóruns de tamanho próximo a $\sqrt{n}$, sendo n o número de processos no sistema. No entanto, a solução não pode ser aplicada para qualquer valor de n. Agrawal e El-Abbadi (1991) utilizam árvores binárias para criar quóruns de tamanho $\log_2 n$, mas que podem chegar a $\lceil (n+1)/2 \rceil$ em cenários com falhas. Além disso, a carga é altamente concentrada na raiz e proporcionalmente nos nodos próximos a ela.

A solução de Thomas faz uso dos chamados quóruns majoritários. Um quórum majoritário Q em P tem $|Q| = \lceil (n+1)/2 \rceil$ elementos. A grande vantagem deste tipo de quórum é a alta disponibilidade. Mesmo em uma solução estática, os quóruns majoritários toleram $f < n/2$ processos falhos [Lipcon 2012]. A tolerância a falhas pode ser ainda aprimorada se o modelo permite a atualização dinâmica do sistema [Amir e Wool 1996]. Além disso, embora outras soluções apresentem modelos teóricos com propriedades otimizadas, as soluções majoritárias continuam sendo as mais utilizadas na prática.

Este trabalho apresenta um sistema de quórum majoritário construído sobre uma topologia de hipercubo virtual chamada VCube. Assume-se um sistema representado por um grafo completo no qual os enlaces são confiáveis. VCube organiza os processos do sistema em clusters progressivamente maiores formando um hipercubo quando não há processos falhos. Processos podem falhar por *crash* e uma falha é permanente. Após uma falha, a topologia do VCube é reestruturada estabelecendo-se novos enlaces entre os processos anteriormente conectados ao processo falho. As propriedades logarítmicas herdadas do hipercubo são mantidas mesmo após a ocorrência de falhas. A avaliação teórica e testes comparativos mostram que os quóruns formados possuem tamanho e carga balanceados e o sistema tolera até $n - 1$ processos falhos.

O restante do texto está organizado nas seguintes seções. A Seção 2 discute os trabalhos correlatos. A Seção 3 apresenta as definições básicas e o modelo do sistema. A Seção 4 descreve o algoritmo de geração de quóruns proposto. Uma avaliação experimental é apresentada na Seção 5. A Seção 6 apresenta a conclusão e os trabalhos futuros.

2. Trabalhos Correlatos

Desde as soluções baseadas em votação propostas por Thomas (1979) e Gifford (1979), muitos outros modelos para a construção de quóruns têm sido apresentados na literatura. Nos modelos estáticos, isto é, aqueles em que a quantidade de processos é fixa e conhecida, o tratamento de falhas normalmente é baseado na substituição de elementos falhos do quórum ou na redefinição dos conjuntos após a falha.

Barbara e Garcia-Molina (1986) propuseram algumas heurísticas para diminuir a vulnerabilidade dos sistemas baseados em votação quando há possibilidade de falhas de processo ou de enlace. Em Barbara et al. (1989) uma solução autonômica de redistribuição de votos é apresentada para o problema da exclusão mútua. Após a ocorrência de falhas, mesmo em caso de particionamento, os votos do processo falho são redistribuídos entre os processos corretos que fazem parte do grupo que contém a maioria.

Maekawa (1985) utilizou planos projetivos finitos para criar quóruns com $c\sqrt{n}$ elementos, $3 \leq c \leq 5$. Planos projetivos são estruturas geométricas derivadas do plano nas quais quaisquer duas linhas se intersectam em apenas um ponto. Entretanto, estes planos existem somente se $n = p^i$, sendo p um número primo e i um inteiro. Para os casos em que não há plano, duas outras soluções são propostas pelo autor, mas com desempenho não balanceado e quóruns de tamanho maior que $\sqrt{n}$. Uma delas organiza o sistema em uma grade de tamanho $l \times l$ com l^2 elementos, na qual cada quórum q_i é composto por todos os elementos da linha e da coluna que cruzam em i. Cada quórum possui $2l - 1$ integrantes e cada par de quóruns possui uma intersecção de, no mínimo, dois elementos. Kumar e Agarwal (2011) apresentaram um generalização para grades multidimensionais que permite a reconfiguração dos quóruns, visando aumentar a disponibilidade do sistema.

Uma solução semelhante às grades organiza os elementos do sistema em um triângulo de tamanho $r \times r$, $r \geq 2$. A quantidade de elementos de cada linha aumenta de 1 até r. Cada quórum é formado por uma linha i completa e por um representante de cada linha $j > i$. Preguiça e Martins (2001) propuseram um sistema de quórum hierárquico com base nesta arquitetura. Cada quórum é formado pela divisão do triângulo em subtriângulos e pela união dos quóruns de cada unidade. Agrawal e El-Abbadi (1991) utilizam uma árvore binária lógica para criar quóruns de tamanho $\lceil \log_2 n \rceil$ a $\lceil (n+1)/2 \rceil$. Cada

quórum é formado por um caminho da raiz até uma folha. Se um nodo neste caminho está falho, o quórum é construído pela junção de um caminho da sub-árvore direita com um caminho da sub-árvore esquerda do nodo falho. No pior caso, um quórum majoritário é formado. Considerando que cada caminho termina em um nodo folha, um quórum não pode ser formado se o caminho termina em um nodo falho. Neste caso o algoritmo retorna uma condição de erro. Uma generalização desta solução foi apresentada por Agrawal e El-Abbadi (1992) para permitir a construção de quóruns com base em árvores arbitrárias.

Uma abordagem híbrida utilizando árvore e grade é apresentada em [Choi e Youn 2012]. Para aumentar a disponibilidade do sistema, a altura da árvore, a quantidade de descendentes e a profundidade da grade são ajustados dinamicamente.

Guerraoui e Vukolić (2010) introduziram a noção de sistemas de quóruns refinados (*refined quorum systems*). Neste modelo, três classes de quóruns são definidas. Os quóruns de primeira classe tem uma grande intersecção com os outros quóruns. Os quóruns de segunda classe tem uma menor intersecção com os quóruns de terceira classe. Estes por sua vez, correspondem aos quóruns tradicionais.

Malkhi et al. (2001) apresentam o conceito de sistema de quórum probabilístico. Em um sistema com n processos, um quórum probabilístico contém $k\sqrt{n}$ processos escolhidos de maneira aleatória e uniforme, sendo k o parâmetro de confiabilidade. De forma semelhante, Abraham (2005) aplicou quóruns probabilísticos em sistemas dinâmicos. De acordo com o autor, quóruns probabilísticos otimizam a carga e a disponibilidade, flexibilizando a propriedade de intersecção, isto é, os quóruns se intersectam com uma probabilidade de acordo com uma determinada distribuição.

3. Definições, Modelo do Sistema e a Topologia Virtual VCube

3.1. Definições

Um sistema distribuído consiste de um conjunto finito P com $n > 1$ processos $\{p_0, .., p_{n-1}\}$. Cada processo executa uma tarefa e está alocado em um nodo distinto. Assim, os termos *nodo* e *processo* são utilizados com o mesmo sentido.

Definição 1 (Quorum)**.** *Seja $P = \{p_0, .., p_{n-1}\}$ um sistema distribuído com $n \geq 1$ processos e $Q = \{q_1, .., q_m\}$ um conjunto de grupos de processos de P tal que $Q \neq \emptyset$, e $Q \subseteq P$. Q é um sistema de quórum se, e somente se, $\forall q_i, q_j \in Q$, $q_i \cap q_j \neq \emptyset$ (propriedade de intersecção).*

Definição 2 (*Coterie*)**.** *Um quórum Q é uma coterie em P se, e somente se, $\forall i, j$ não existe $q_i, q_j \in Q$ tal que $q_i \subset q_j$ (propriedade de minimalidade).*

O modelo de *coteries* garante que existem apenas conjuntos mínimos no sistema, isto é, o tamanho de qualquer conjunto é mínimo porque não existe nenhum conjunto menor contido nele.

O **tamanho** do quórum é a quantidade de processos contida nele. A **carga** está relacionada ao número de quóruns em que cada processo está presente. Um sistema de quóruns é dito s-uniforme se cada quórum possui exatamente s elementos. A carga está relacionada ainda à qualidade do sistema de quórum e pode ser associada tanto aos processos individualmente quanto ao sistema como um todo.

Em termos de **disponibilidade**, duas medidas são empregadas aos quóruns: resiliência e probabilidade de falhas. Um quórum é t-resiliente se, mesmo após a falha

de t processos quaisquer, existe ao menos um quórum $q \in Q$ tal que nenhum processo pertencente a q falhou. Neste caso, ao menos um quórum sempre poderá ser ativado. A resiliência máxima é $\lfloor (n-1)/2 \rfloor$, proveniente dos quóruns majoritários. A probabilidade de falhas é a probabilidade de que ao menos um processo em cada quórum esteja falho, isto é, nenhum quórum possui um processo sem falha [Naor e Wool 1998]. De acordo com a propriedade de intersecção, se todos os processos em um dado quórum falham, ao menos um processo falho estará presente em todos os demais quóruns. No entanto, estes limites são especificados para quóruns estáticos, isto é, aqueles que não se adaptam após a ocorrência de falhas.

3.2. Modelo do Sistema

Comunicação. Os processos se comunicam através do envio e recebimento de mensagens. A rede é representada por um grafo completo, isto é, cada par de processos está conectado diretamente por enlaces ponto-a-ponto bidirecionais. No entanto, processos são organizados em uma topologia de hipercubo virtual, chamada VCube. Em um hipercubo de d dimensões (d-cubo) cada processo é identificado por um endereço binário i_{d-1}, i_{d-2}, i_0. Dois processos estão conectados se seus endereços diferem em apenas um bit. Se não existem processos falhos, VCube é um hipercubo completo. Após uma falha, a topologia do VCube é modificada dinamicamente. O VCube é descrito com mais detalhes na Seção 3.3 abaixo. As operações de envio e recebimento são atômicas. Enlaces são confiáveis, garantindo que as mensagens trocadas entre dois processos nunca são perdidas, corrompidas ou duplicadas. Falhas de particionamento da rede não são tratadas.

Modelo de Falhas. O sistema admite falhas do tipo *crash* permanente. Um processo que nunca falha é considerado *correto* ou *sem-falha*. Caso contrário ele é considerado *falho*. Falhas são detectadas por um serviço de detecção de falhas perfeito, isto é, nenhum processo é detectado como falho antes do início da falha e todo processo falho é detectado por todos os processos corretos em um tempo finito [Freiling et al. 2011].

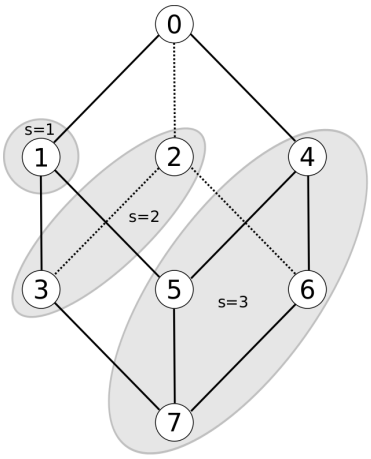
3.3. A Topologia Virtual VCube

O hipercubo virtual utilizado neste trabalho, denominada VCube, é criado e mantido com base nas informações de diagnóstico obtidas por meio de um sistema de monitoramento de processos descrito em Ruoso (2013). Cada processo que executa o VCube é capaz de testar outros processos no sistema para verificar se estão corretos ou falhos. Para isso, o processo executa um procedimento de teste e aguarda por uma resposta. Um processo é considerado correto ou sem-falha se a resposta ao teste for recebida corretamente dentro do intervalo de tempo esperado. 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. Os testes são executados em rodadas. Para cada rodada um processo i testa o primeiro processo sem-falha j na lista de processos de cada cluster s e obtém informação sobre os processos naquele cluster.

Os membros de cada cluster s e a ordem na qual eles são testados por um processo i são dados pela lista $c_{i,s}$, definida a seguir. O símbolo $\oplus$ representa a operação binária de OU exclusivo (XOR):

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

A Figura 1 exemplifica a organização hierárquica dos processos em um hipercubo de três dimensões com $n = 8$ 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 do processo p_1. Em seguida, p_0 testa o processo p_2, que é primeiro processo no cluster $c_{0,2} = (2, 3)$, e obtém informações sobre p_2 e p_3. Por fim, p_0 executa testes no processo p_4 do cluster $c_{0,3} = (4, 5, 6, 7)$ e obtém informações sobre os processos p_4, p_5, p_6 e p_7.



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.

4. O Sistema de Quórum Proposto

Esta seção apresenta a abordagem para construção dos clusters hierárquicos baseados na topologia virtual em hipercubo do VCube. Inicialmente são definidas algumas funções e, em seguida, o algoritmo proposto.

4.1. Definição das Funções

Com base na organização virtual dos processos utilizada pelo VCube e na função $c_{i,s}$ apresentados na Seção 3.3, foram definidas algumas funções para auxiliar a implementação do algoritmo de quóruns proposto neste trabalho.

Inicialmente, o conjunto $correct_i$ é definido para registrar as informações que o processo i tem sobre o estado dos processos informadas pelo VCube. Estas informações são obtidas através dos testes realizados pelo monitoramento. Sendo assim, em razão da latência do detector, é possível que um processo j falho ainda seja considerado como correto pelo processo i. No entanto, tão logo o processo i seja informado sobre a falha, j é removido de $correct_i$ e, após o diagnóstico completo, todos os processos terão a mesma informação sobre processos corretos e falhos.

Seja o resultado da função $c_{i,s}$ uma lista $(a_1, a_2, .., a_m)$, $m = 2^{s-1}$ dos processos do cluster s em relação ao processo i. A lista de m' elementos considerados corretos por i em um cluster $c_{j,s}$ é definida como $FF_cluster_i(s) = (b_1, .., b_{m'}), b_k \in (c_{i,s} \cap correct_i)$, $k = 1..m'$, $m' \leq m$. Desta forma, a função $FF_mid_i(s) = (b_1, .., b_{\lceil m'/2 \rceil})$ gera um conjunto que contém a metade absoluta dos processos considerados corretos pelo processo i em um cluster $c_{i,s}$. Se não existem processos corretos no cluster s, $FF_cluster_i(s) = \perp$ e, por conseguinte, $FF_mid_i(s) = \perp$. Note que esta função é dependente do conhecimento

atual que um processo tem a respeito das falhas no sistema. Devido à latência de detecção, em um mesmo espaço de tempo, é possível que dois processos possuam visões diferentes sobre quais processos estão corretos ou falhos.

4.2. Descrição do Algoritmo

Em termos gerais, cada processo i constrói o seu próprio quórum adicionando a si mesmo e a metade absoluta dos elementos j que ele considera sem-falha ($j \in correct_i$) em cada cluster $c_{i,s}$ do hipercubo virtual definido por VCube. O estado dos processos é informado pelo VCube. O Algoritmo 1 apresenta um pseudo-código desta solução.

Algoritmo 1 Obtenção do quórum de um processo j pelo processo i

1: $correct_i \leftarrow \{0, .., n-1\}$
2: **function** GETQUORUM
3: $\quad q_i \leftarrow \{i\}$
4: $\quad$ **for** $s \leftarrow 1, .., \log_2 n$ **do**
5: $\quad\quad q_i \leftarrow q_i \cup FF_mid_i(s)$
6: $\quad$ **return** q_i
7: **upon notifying** CRASH(j)
8: $\quad correct_i \leftarrow correct_i \setminus \{j\}$

Como exemplo, considere o cenário sem falhas representado pelo hipercubo de três dimensões da Figura 2(a), que representa os quóruns dos processos p_0 e p_7. Estes processos foram escolhidos por estarem o mais distante possível um do outro no hipercubo. O quórum q_0 calculado pelo processo p_0 é composto por ele mesmo e pela metade absoluta dos clusters $c_{0,1} = (1)$, $c_{0,2} = (2, 3)$ e $c_{0,3} = (4, 5, 6, 7)$. Neste caso, $FF_mid_0(1) = (1)$, $FF_mid_0(2) = (2)$ e $FF_mid_0(3) = (4, 5)$. Logo, $q_0 = \{0, 1, 2, 4, 5\}$. O quórum q_7 referente ao processo p_7 é formado por $FF_mid_7(1) = (6)$, $FF_mid_7(2) = (5)$ e $FF_mid_7(3) = (3, 2)$. Assim, $q_7 = \{7, 6, 5, 3, 2\}$. Logo, a intersecção, $q_0 \cap q_7 = \{2, 5\}$. A Figura 2(b) ilustra um cenário após a falha dos processos da interseção p_2 e p_5. Neste caso, a topologia virtual é reestruturada e o processo p_6 passa a fazer parte do quórum de p_0 pois, dada a falha de p_5, $FF_mid_0(3) = (4, 6)$. O processo p_2 é incluído no quórum de p_7 por razões semelhantes. Assim, p_1 e p_6 passam a compor a nova interseção.

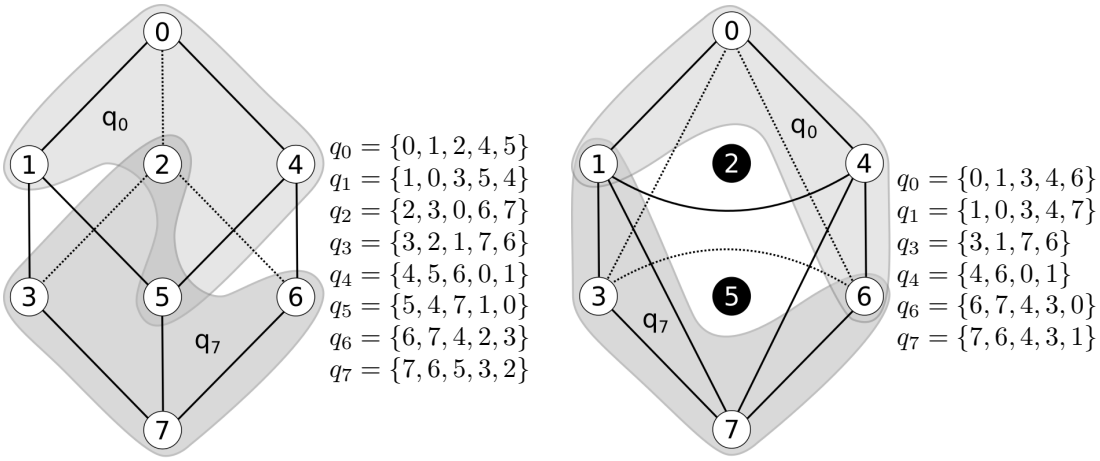
Teorema 1 (Intersecção)**.** *Todo quórum q_i construído por um processo i utilizando o Algoritmo 1 tem uma interseção com cada outro quórum do sistema de, no mínimo, dois elementos.*

Prova. A prova é por indução baseada nos clusters do VCube.

Base: o teorema é valido para um VCube de dimensão 1 com dois processos $\{p_0, p_1\}$. O processo p_0 adiciona a si mesmo e o processo p_1 ao seu quórum, visto que $c_{0,1} = (1)$ e $FF_mid_0(1) = (1)$. O processo p_1 adiciona p_0 ao seu quórum de forma semelhante. Neste caso, $q_0 \cap q_1 = \{0, 1\}$.

Hipótese: suponha que o teorema é válido para um VCube de d dimensões.

Passo: considere um VCube com $d + 1$ dimensões com $n = 2^{d+1}$ processos. Cada cluster $c_{i,s}$ tem $2^s/2$ elementos. A soma dos elementos de todos os clusters s de um processo i é



(a) Sem falhas (b) Processos p_2 e p_5 falhos

Figura 2. Quóruns do VCube de 3 dimensões com representação gráfica para os processos p_0 e p_7.

dada por $2^1/2 + .. + 2^d/2 + 2^{d+1}/2 = 2^{d+1} - 1 = n - 1$. Portanto, a soma das metades absolutas dos elementos de cada cluster s é dada por:

$$\left\lceil \frac{\frac{2^1}{2} + .. + \frac{2^d}{2} + \frac{2^{d+1}}{2}}{2} \right\rceil = \left\lceil \frac{2^{d+1} - 1}{2} \right\rceil = \left\lceil \frac{n-1}{2} \right\rceil$$

Sendo n uma potência de 2 e, portanto, par, $\left\lceil \frac{n-1}{2} \right\rceil$ equivale a $\frac{n}{2}$.

Sejam q_i e q_j dois quóruns construídos por dois processos quaisquer i e j, respectivamente. Como cada processo i adiciona a si mesmo ao seu quórum q_i, $|q_i| = 1 + \frac{n}{2}$. Seja $P = \{p_0, .., p_{n-1}\}$ o conjunto de processos que representa o VCube em questão. Considere que $q_i = \{p_0, .., p_{n/2-1}\}$ e $q_j = \{p_{n/2}, .., p_{n-1}\}$ contém cada um, a metade distinta dos elementos de P. A adição de qualquer elemento $p_a \in q_i$ a q_j e de $p_b \in q_j$ a p_i garante a maioria para ambos e, por conseguinte, a intersecção em, no mínimo, dois elementos: $\{p_a, p_b\}$.

Assim, por indução, o teorema é válido. □

4.3. Análise do Algoritmo

Nesta seção são discutidas as propriedades do sistema de quóruns proposto. As métricas avaliadas são o tamanho e a carga dos quóruns, e os aspectos de disponibilidade e tolerância a falhas.

Tamanho e Carga dos Quóruns. Em um sistema de quóruns ideal, todos os quóruns são do mesmo tamanho, a interseção entre dois quóruns quaisquer tem o mesmo número de elementos e cada elemento pertence ao mesmo número de quóruns.

O tamanho de um quórum q_i no VCube pode ser dado por:

$$|q_i| = 1 + \sum_{s=1}^{\log_2 n} |FF_mid_i(s)| \tag{2}$$

Considerando que cada cluster possui 2^s processos, a metade absoluta quando não existem processos falhos no cluster s é $\lceil |c_{i,s}|/2 \rceil$. No total, a metade dos processos é $\sum_{s=1}^{\log_2 n} \lceil \frac{2^s}{2} \rceil$. A inclusão do próprio processo i completa a maioria. Assim, nos cenários sem falha, os quóruns são uniformes e possuem exatamente $(n/2+1)$ elementos. Além disso, nenhum conjunto está contido em outro (*coteries*).

Seja f_s o total de processos falhos em um cluster s. Se $|c_{i,s}| - f_s$ é par, a metade que corresponde aos processos corretos é mantida. Se é ímpar, a quantidade de elementos que fazem parte do quórum do cluster é o resultado da divisão inteira de $\frac{|FF_cluster_i(s)|}{2}$ acrescido de 1. Assim, no pior caso, o tamanho do quórum será:

$$|q_i| = 1 + \sum_{s=1}^{\log_2 n} \left\lfloor \frac{|c_{i,s}| - f_s}{2} \right\rfloor + c, c = \begin{cases} 0 & \text{se } |c_{i,s}| - f_s \text{ é par} \\ 1 & \text{se } |c_{i,s}| - f_s \text{ é ímpar} \end{cases} \quad (3)$$

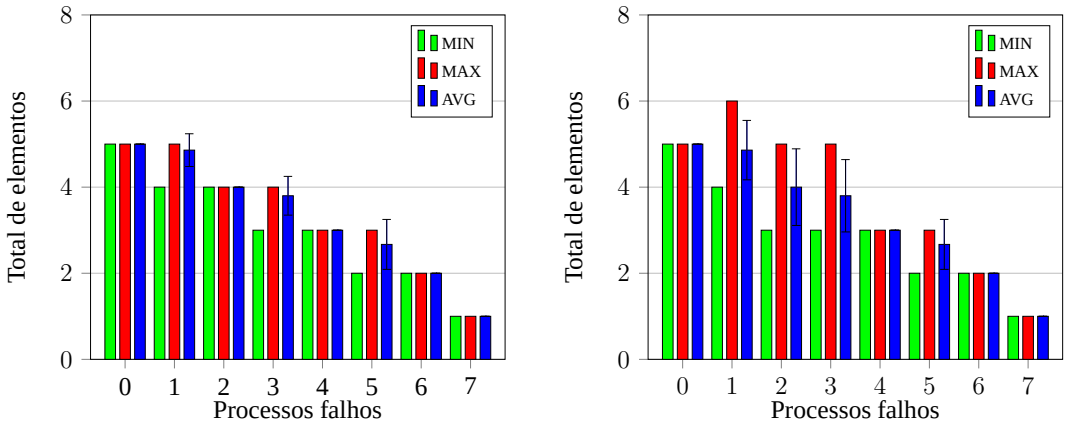
Em relação à carga, quando não existem processos falhos, a característica simétrica da $c_{i,s}$ garante a igual distribuição dos processos nos clusters e, portanto, o equilíbrio na inclusão dos elementos em cada quórum. Em caso de falha de um processo j, se $j \in c_{i,s}$, os processos corretos k que pertencem ao cluster s do processo i terão uma maior probabilidade de pertencer a um novo quórum. Se após a falha $c_{i,s} - f_s$ é ímpar, ao menos um processo k será incluído no quórum do processo i, possivelmente aumentando a carga de k, exceto se k pertencia ao quórum do processo j que falhou.

Como exemplo, considere o sistema P com $n = 8$ processos. O gráfico da Figura 3(a) mostra o tamanho dos quóruns quando não existem processos falhos e após a detecção de falhas. Inicialmente, quando não existem processos falhos cada quórum possui exatamente $(n/2+1) = 5$ elementos. Após a ocorrência de uma falha, restam 7 processos corretos e os quóruns passam a ter entre 4 (a maioria absoluta de 7) e 5 elementos (a maioria mais um). Quando existem mais processos falhos o comportamento é semelhante. A carga está representada na Figura 3(b). Quando não há falhas o tamanho e a carga são idênticos, o que mostra a uniformidade dos quóruns. Após a ocorrência de uma falha, a carga varia entre 4 e 6, mas a média se mantém em 5. Para 4 processos falhos o tamanho e a carga são iguais novamente e cada quórum possui a maioria dos processos corretos, isto é, $4/2+1 = 3$ elementos.

Disponibilidade. A disponibilidade é a capacidade do sistema de quóruns em tolerar falhas. Na solução com VCube, após o diagnóstico completo do nodo falho, todos os processos atualizam seus quóruns automaticamente. Assim, mesmo após $n-1$ falhas é possível reconstruir o sistema de quóruns, ou seja, o sistema proposto é $(n-1)$-resiliente.

5. Avaliação Experimental

Nesta seção são apresentados os resultados dos experimentos comparativos realizados com a solução de quóruns majoritários proposta e com o algoritmo de quóruns em árvore de Agrawal e El-Abbadi (1991), denominada TREE. Foi utilizada uma árvore binária balanceada com raiz em p_0, na qual os demais processos são adicionados sequencialmente seguindo uma distribuição em largura. Primeiro são apresentados os resultados para cenários sem processos falhos e, em seguida, para os cenários com falhas.



(a) Tamanho (b) Carga

Figura 3. Tamanho e carga dos quóruns em um sistema VCube com 8 processos.

5.1. Cenários sem falhas

A Tabela 1 mostra as propriedades dos quóruns TREE e VCube para cenários com número de processos n variando de 8 a 1024 em potência de dois. A quantidade de quóruns em cada algoritmo está representada por $|Q|$. Para o tamanho dos quóruns e a carga de um processo, isto é, o total de elementos em cada quórum e a quantidade de quóruns a qual cada elemento pertence, são mostrados o menor ($<$), o maior ($>$), a média ($\overline{q_i}$) e o desvio padrão (σ).

Quando não há falhas, o algoritmo TREE gera $|Q| = n/2$ quóruns com tamanho médio $\overline{q_i}$ muito próximo a $\log_2 n$. Neste exemplo, em função do valor de n, a árvore gerada é completa e balanceada, exceto pelo último elemento folha que compõe o último nível da árvore. Este elemento faz parte do único quórum formado pelo ramo esquerdo mais longo da árvore e que possui tamanho $\log_2 n + 1$. A solução VCube, por outro lado, gera n quóruns, um para cada processo, e cada quórum possui $n/2 + 1$ elementos. O tamanho e a carga são idênticos, visto que, em função da simetria do cálculo de $c_{i,s}$, cada processo está contido em $n/2 + 1$ quóruns.

Tabela 1. Resultados dos testes em cenários sem falhas.

	TREE	Tamanho				Carga				VCube	Tamanho/
n	$\lvert Q\rvert$	$<$	$>$	$\overline{q_i}$	σ	$<$	$>$	$\overline{q_i}$	σ	$\lvert Q\rvert$	Carga
8	4	3	4	3,25	0,50	1	4	1,63	1,06	8	5
16	8	4	5	4,13	0,35	1	8	2,06	1,88	16	9
32	16	5	6	5,06	0,25	1	16	2,53	3,07	32	17
64	32	6	7	6,03	0,18	1	32	3,02	4,77	64	33
128	64	7	8	7,02	0,13	1	64	3,51	7,18	128	65
256	128	8	9	8,01	0,09	1	128	4,00	10,58	256	129
512	256	9	10	9,00	0,06	1	256	4,50	15,35	512	257
1024	512	10	11	10,00	0,04	1	512	5,00	22,07	1024	513

Em relação à carga, verifica-se que à medida que o VCube a distribui igualitariamente entre os processos, TREE sobrecarrega a raiz, bem como proporcionalmente os nodos nos níveis mais próximos a ela. Considerando uma árvore binária, a raiz está contida em todos os quóruns, os elementos do nível seguinte em $|Q|/2$ conjuntos e assim

sucessivamente até as folhas, que pertencem a apenas um quórum. Nas medidas de carga da Tabela 1, a maior carga ($>$) é da raiz é a menor ($<$) representa as folhas. O desvio padrão confirma esta dispersão. Além disso, em um sistema que utiliza TREE, se todos os processos precisam acessar um quórum a carga pode ser ainda maior, visto que $|Q|$ representa metade do número de processos.

5.2. Cenários com falhas

Inicialmente foi testado o impacto da falha de um único processo nas duas soluções. A Tabela 2 mostra os valores de tamanho e carga dos quóruns no VCube quando um único processo está falho. Nota-se que o tamanho dos quóruns se mantém estável e a carga, embora não mais simétrica, é ainda bem distribuída.

Tabela 2. Resultados dos testes com VCube para a falha de um único processo.

VCube		Tamanho				Carga			
n	$\|Q\|$	$<$	$>$	$\overline{q_i}$	σ	$<$	$>$	$\overline{q_i}$	σ
8	7	4	5	4,86	0,38	4	6	4,86	0,69
16	15	8	9	8,93	0,26	8	10	8,93	0,70
32	31	16	17	16,97	0,18	16	18	16,97	0,71
64	63	32	33	32,98	0,13	32	34	32,98	0,71
128	127	64	65	64,99	0,09	64	66	64,99	0,71
256	255	128	129	129,00	0,06	128	130	129,00	0,71
512	511	256	257	257,00	0,04	256	258	257,00	0,71
1024	1023	512	513	513,00	0,03	512	514	513,00	0,71

A posição do nodo falho no VCube influencia os quóruns independentemente, não tem impacto no desempenho geral do sistema. Em TREE, por outro lado, cada falha em um nível da árvore altera o sistema como um todo, exceto pelos elementos folha, que invalidam o quórum nos caminhos da árvore que os contém e nos quais o impacto é maior quando combinados com múltiplas falhas. A Tabela 3 mostra os resultados para um único processo falho no segundo nível da árvore, isto é, um filho da raiz. O tamanho dos quóruns aumenta em relação ao cenário sem falhas, especialmente para os sistemas com maior número de processos. Isto é esperado em função do maior número de combinações entre as sub-árvores esquerda e direita do nodo falho. A carga também é aumentada, mas a carga mínima ainda é 1, presente nos quóruns da sub-árvore esquerda da raiz.

Tabela 3. Resultados dos testes com TREE para um processo falho não-raiz (p_1).

TREE		Tamanho				Carga			
n	$\|Q\|$	$<$	$>$	$\overline{q_i}$	σ	$<$	$>$	$\overline{q_i}$	σ
8	3	3	4	3,33	0,58	1	3	1,43	0,79
16	8	4	6	4,75	0,89	1	8	2,53	1,85
32	24	5	8	6,50	1,14	1	24	5,03	5,24
64	80	6	10	8,50	1,29	1	80	10,79	15,54
128	288	7	12	10,61	1,30	1	288	24,06	45,91
256	1088	8	14	12,74	1,20	1	1088	54,34	134,12
512	4224	9	16	14,83	1,04	1	4224	122,61	388,01
1024	16640	10	18	16,90	0,87	1	16640	274,89	1114,43

A Tabela 4, também com resultados obtidos para TREE, apresenta os dados para o caso em que a raiz (processo p_0) está falha. Em função das combinações entre os elementos da árvore esquerda e direita da raiz, a quantidade de quóruns disponíveis representada

na coluna $|Q|$ aumenta consideravelmente. O tamanho dos quóruns praticamente dobrou em relação ao cenário sem falhas. As cargas mínima e máxima também aumentam, mas continuam muito desproporcionais. Como cada quórum é formado pela combinação de um caminho da sub-árvore esquerda com um caminho da sub-árvore direita, a carga mínima aumenta proporcionalmente ao número de possíveis combinações.

Tabela 4. Resultados dos testes com TREE para a falha da raiz (p_0).

TREE		Tamanho				Carga			
n	$\|Q\|$	$<$	$>$	$\overline{q_i}$	σ	$<$	$>$	$\overline{q_i}$	σ
8	4	4	5	4,50	0,58	2	4	2,57	0,98
16	16	6	7	6,25	0,45	4	16	6,67	4,19
32	64	8	9	8,13	0,33	8	64	16,77	14,95
64	256	10	11	10,06	0,24	16	256	40,89	49,00
128	1024	12	13	12,03	0,17	32	1024	97,01	152,61
256	4096	14	15	14,02	0,12	64	4096	225,13	449,65
512	16384	16	17	16,01	0,09	128	16384	513,25	1353,97
1024	65536	18	19	18,00	0,06	256	65536	1153,48	3930,10

Em resumo, embora o tamanho dos quóruns de VCube seja maior que TREE, a carga de VCube é uniformemente distribuída e o sistema é mais estável em cenários com falhas.

6. Conclusão

Este trabalho apresentou uma solução distribuída e autonômica para a construção de quóruns majoritários em sistemas distribuídos sujeitos a falhas de *crash*. Os quóruns são construídos e mantidos sobre uma topologia de hipercubo virtual denominada VCube. O VCube organiza os processos em clusters progressivamente maiores e os conecta através de enlaces confiáveis de forma que, se não há falhas, um hipercubo é formado. Cada processo constrói seu próprio quórum adicionado a si mesmo e a metade absoluta dos processos sem falha em cada cluster.

Os sistema de quóruns com VCube foi comparado com um modelo em árvore binária denominado TREE. Os experimentos em cenários sem falhas mostram que os quóruns gerados no VCube são uniformes e têm tamanho $\lceil (n+1)/2 \rceil$. Nos cenários com falha, o tamanho e a carga dos quóruns pode aumentar, mas ainda se mantém equilibrados. Além disso, ao contrário de TREE, a posição do processo falho não tem impacto no desempenho geral do sistema. Em relação à disponibilidade, o VCube é capaz de reconstruir os quóruns com até $n-1$ processos falhos.

Como trabalhos futuros, uma aplicação será implementada para o problema da exclusão mútua distribuída baseada em quóruns. Um algoritmo de *multicast* para o quórum no VCube está em desenvolvimento.

Agradecimentos

Estre trabalho teve apoio da Fundação Araucária/SETI (convênio 341/10, proj. 19.836) e do CNPq, proj. 309143/2012-8.

Referências

Abawajy, J. e Mat Deris, M. (2013). Data replication approach with data consistency guarantee for data grid. *IEEE Trans. Comput.*, PP(99):1–1.

Abraham, I. e Malkhi, D. (2005). Probabilistic quorums for dynamic systems. *Distrib. Comput.*, 18(2):113–124.

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

Agrawal, D. e El Abbadi, A. (1992). The generalized tree quorum protocol: an efficient approach for managing replicated data. *ACM Trans. Database Syst.*, 17(4):689–717.

Amir, Y. e Wool, A. (1996). Evaluating quorum systems over the internet. In *FTCS*, pages 26–, Washington, DC, USA. IEEE Computer Society.

Atreya, R., Mittal, N. e Peri, S. (2007). A quorum-based group mutual exclusion algorithm for a distributed system with dynamic group set. *IEEE Trans. Parallel Distrib. Syst.*, 18(10):1345–1360.

Barbara, D. e Garcia-Molina, H. (1986). The vulnerability of vote assignments. *ACM Trans. Comput. Syst.*, 4(3):187–213.

Barbara, D., Garcia-Molina, H. e Spauster, A. (1989). Increasing availability under mutual exclusion constraints with dynamic vote reassignment. *ACM Trans. Comput. Syst.*, 7(4):394–426.

Choi, S. C. e Youn, H. Y. (2012). Dynamic hybrid replication effectively combining tree and grid topology. *J. Supercomput.*, 59(3):1289–1311.

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

Fujita, S. (1998). A quorum based k-mutual exclusion by weighted k-quorum systems. *Inf. Process. Lett.*, 67(4):191–197.

Garcia-Molina, H. e Barbara, D. (1985). How to assign votes in a distributed system. *J. ACM*, 32(4):841–860.

Gifford, D. K. (1979). Weighted voting for replicated data. In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, SOSP '79, pages 150–162, New York, NY, USA. ACM.

Guerraoui, R. e Vukolić, M. (2010). Refined quorum systems. *Distributed Computing*, 23(1):1–42.

Kumar, V. e Agarwal, A. (2011). Generalized grid quorum consensus for replica control protocol. In *Computational Intelligence and Communication Networks (CICN), 2011 International Conference on*, pages 395–400.

Lipcon, T. (2012). Quorum-based journaling in CDH4.1. Disponível em: `http://blog.cloudera.com/blog/2012/10/quorum-based-journaling-in-cdh4-1/`. Acessado em: 12/02/2014.

Liskov, B. e Rodrigues, R. (2006). Tolerating byzantine faulty clients in a quorum system. In *ICDCS '06*, pages 34–, Washington, DC, USA. IEEE Computer Society.

Liu, T.-J., Wang, W.-C. e Tseng, C.-M. (2011). Organize metadata servers by using quorum system. In *IEEE/SICE Int'l Symp. Syst. Integration*, pages 1125–1130.

Maekawa, M. (1985). A $\sqrt{n}$ algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst.*, 3(2):145–159.

Malkhi, D., Reiter, M. K., Wool, A. e Wright, R. N. (2001). Probabilistic quorum systems. *Information and Computation*, 170(2):184–206.

Merideth, M. e Reiter, M. (2008). Write markers for probabilistic quorum systems. In *Principles of Distributed Systems*, volume 5401 of *LNCS*, pages 5–21. Springer Berlin Heidelberg.

Merideth, M. G. e Reiter, M. K. (2007). Probabilistic opaque quorum systems. In *DISC'07*, pages 403–419.

Merideth, M. G. e Reiter, M. K. (2010). Selected results from the latest decade of quorum systems research. In *Replication*, volume 5959 of *LNCS*, pages 185–206. Springer-Verlag, Berlin, Heidelberg.

Naimi, M. e Thiare, O. (2013). A distributed deadlock free quorum based algorithm for mutual exclusion. *IJCSIS*, 11(8):7–13.

Naor, M. e Wool, A. (1996). Access control and signatures via quorum secret sharing. In *ACM Conference on Computer and Communications Security*, pages 157–168. ACM.

Naor, M. e Wool, A. (1998). The load, capacity, and availability of quorum systems. *SIAM J. Comput.*, 27(2):423–447.

Preguica, N. e Martins, J. L. (2001). Revisiting hierarchical quorum systems. In *Proc. Int'l Conf. Distr. Comp. Syst. (ICDCS-21)*.

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

Ruoso, V. K. (2013). Uma estratégia de testes logarítmica para o algoritmo Hi-ADSD. Master's thesis, Universidade Federal do Paraná.

Tanaka, K., Higaki, H. e Takizawa, M. (1999). Quorum-based protocol for group communication. In *Proc. Int'l Workshops on Parallel Processing*, pages 24–29.

Thomas, R. H. (1979). A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209.

Vukolić, M. (2010). The origin of quorum systems. *Bulletin of the EATCS*, 101:125–147.

Potência do Sinal de Recepção como Suporte à Detecção de Mobilidade em Detectores de Defeitos

Miguel A. Baggio, Raul Ceretta Nunes, Marcia Pasin, Antônio Rodrigo D. de Vit

Programa de Pós-Graduação em Informática (PPGI)
Universidade Federal de Santa Maria (UFSM)
Av. Roraima 1000 - Cidade Universitária - 97105-900 – RS – Brasil

{baggio, ceretta, marcia}@inf.ufsm.br; rodrigodevit@smail.ufsm.br

Abstract. *Failure detector is a building block component in reliable distributed systems and its design depends strongly on the model of the distributed system. This dependence has demanded new algorithms to address nodes movement on mobile ad hoc networks (MANETs). This paper presents a new unreliable gossip-based failure detector that differentiates faulty and mobile nodes. Our approach explores the information about signal power of received messages mapped into a little locality region history. The experimental results have shown quality of service improvements when compared with the traditional gossiping algorithm.*

Resumo. *Detector de defeitos é um componente essencial na construção de sistemas distribuídos confiáveis e seu projeto depende fortemente do modelo de sistema distribuído, o que tem demandado soluções para tratar a movimentação de nós em redes móveis ad hoc (MANETs). Este trabalho apresenta um detector de defeitos assíncrono não-confiável baseado em fofoca que diferencia nós defeituosos e móveis através da manutenção de informações sobre a potência do sinal de recepção nos nós do sistema mapeadas em um pequeno histórico de regiões. As avaliações demonstram melhoras na qualidade de serviço do detector quando comparadas com o algoritmo de fofoca tradicional.*

1. Introdução

Algoritmos para detecção de defeitos, um bloco de construção básico para a construção de sistemas distribuídos confiáveis, têm sido amplamente estudados para operação em redes móveis sem fio [Hutle 2004][Friedman e Tcharny 2005][Sridhar 2006][Sens *et al*. 2008] [Mahapatro e Khilar 2013]. Entretanto, ainda persiste o desafio de diferenciar precisamente a mobilidade de um nó de um estado defeituoso.

Especialmente em MANETs (*Mobile Ad hoc NETworks*) [Loo *et al*. 2012], uma rede auto-configurável (sem uma administração centralizada) em que os nós caracterizam-se por serem móveis e possuírem conexões dinâmicas, diferenciar uma desconexão por movimentação de uma desconexão por falha do nó exige um algoritmo que permita classificar um nó como móvel, descartando assim a possibilidade de falha. Em MANETS, os nós comunicam-se com seus vizinhos por mensagens *broadcast* via rádio, limitado ao alcance de transmissão de seus rádios. Para um detector de defeitos a característica de mobilidade dos nós e a limitação no alcance de transmissão podem

ocasionar falsas suspeitas (imprecisão), dado que um nó p_i pode suspeitar erroneamente que um nó p_j esteja falho quando, de fato, p_j pode apenas ter se movimentado para uma região onde seu alcance de transmissão não alcança p_i.

De acordo com Gracioli e Nunes (2007), para não depender de hierarquia ou topologia fixa, grande parte das estratégias de detecção de defeitos para redes móveis sem fio tem como base o algoritmo *Gossip* [Renesse *et al.* 1998]. O *Gossip* é um algoritmo epidêmico baseado no fenômeno social chamado fofoca, onde um nó difunde (por *broadcast* ou *multicast*) para seus vizinhos locais (dentro do seu alcance de transmissão) informação sobre um grupo de nós vizinhos (locais ou remotos). Neste algoritmo, quando um nó detector de defeitos fica muito tempo sem receber notícias sobre um determinado nó (equivalente a um tempo nomeado *Tcleanup*), o detector suspeita do nó. O ponto chave do algoritmo é que um nó conhece seus vizinhos ao longo do tempo através de mensagens que passam de vizinhos para vizinhos.

Como a estratégica epidêmica por si só não permite diferenciar nós defeituosos de nós móveis, com base nela foram exploradas técnicas de contagem de *hops* [Friedman e Tcharny, 2005], de determinação da variação do atraso de comunicação [Hutle, 2004], de inclusão de novos *rounds* de comunicação [Zia *et al.*, 2009] e de difusão de listas de informações [Sridhar, 2006]. Adicionalmente, Sens *et al*. (2008) propuseram uma técnica que independe de temporizador e que habilita a detecção apenas com base em informações sobre falhas. Diferentemente dos demais, este artigo explora a característica de variabilidade de potência nas transmissões de uma rede MANET [Gomez 2007] e propõe uma nova técnica de detecção de defeitos para redes móveis sem fio: o uso da medida de potência do sinal de recepção como elemento classificador da mobilidade de um nó. Classificando os nós em regiões, a técnica proposta permite identificar a movimentação de um nó e gerar uma informação de previsão de movimentação do nó, a qual é utilizada pelo detector de defeitos para se ajustar à mobilidade e melhorar a qualidade da detecção de defeitos. A solução proposta foi avaliada em um ambiente de simulação para MANETs e demonstrou melhora na qualidade de serviço do detector.

O restante deste trabalho está organizado como segue. A Seção 2 apresenta o modelo de sistema e discute detectores de defeitos. A Seção 3 apresenta definições e conceitos usados ao longo deste artigo. A Seção 4 detalha a proposta de detector de defeitos que usa a potência do sinal de recepção como parâmetro de classificação no suporte à mobilidade. A Seção 5 apresenta os resultados práticos da simulação do algoritmo. A Seção 6 apresenta os trabalhos relacionados e a Seção 7 conclui o trabalho.

2. Modelo de Sistema e Detectores de defeitos

Esta seção apresenta o modelo de sistema distribuído considerado e uma revisão sobre os detectores de defeitos.

2.1 Modelo de Sistema

O sistema distribuído considerado neste trabalho é assíncrono e composto por um conjunto finito de nós móveis $\prod = \{p_1, \ldots, p_n\}$, onde $n > 1$. Cada nó tem sua própria memória, unidade de processamento, relógio local e processos em execução. Não há relógio global. Os nós se comunicam através de troca de mensagens, usando difusão

(*broadcast*) via rádio com alcance de transmissão finito e igual para todos os nós. A potência do sinal de recepção é variável dentro do alcance de transmissão. Uma mensagem *m* enviada por um nó p_i somente é recebida por outro nó p_j que esteja dentro do seu alcance de transmissão quando *m* for enviada e a potência do sinal de recepção da mensagem *m* puder ser identificada por p_j. Todo nó móvel p_i percorre, num tempo finito, pelo menos uma trajetória *T* tal que ao longo de *T* o nó p_i entra no alcance de transmissão de pelo menos um nó p_j e recebe uma mensagem *m* de p_j, e pelo menos um nó p_j recebe uma mensagem de p_i. A topologia de rede é dinâmica e os canais de comunicação são bidirecionais e confiáveis (não alteram, não criam e não perdem mensagens). Os nós são auto-organizáveis e adaptam-se de forma autônoma a falhas do tipo *crash*. Para controlar falhas, cada nó no sistema distribuído executa um serviço de detecção de defeitos.

2.2 Detector de defeitos

Dada a natureza assíncrona do sistema distribuído, o conceito de detectores de defeitos não confiáveis foi introduzido por Chandra e Toueg (1996) para contornar o problema da impossibilidade de resolver o consenso num sistema assíncrono sujeito a falhas [Fischer *et al*. 1985]. A ideia básica é encapsular o problema no detector de defeitos e fornecer uma noção de estado para a aplicação. O estado pode ser expresso por uma lista de nós suspeitos de estarem falhos [Chandra e Toueg, 1996], uma lista de nós confiáveis [Aguilera *et al*. 1998] ou uma lista de contadores de mensagens [Aguilera *et al*. 1997]. Normalmente o estado que é informado à aplicação por um detector de defeitos corresponde à percepção do módulo de detecção local e um nó p_j é dito *suspeito* quando um detector p_i não recebe uma mensagem do nó p_j dentro de um determinado intervalo de tempo.

Há diferentes estratégias de comunicação para construir a noção de estado no detector [Felber *et al*. 1999]. Em redes cabeadas, as mais tradicionais são: *push*, estratégia baseada no envio periódico de mensagens de estado corrente (*I_am_alive!*) ou *heartbeat* [Aguilera *et al*. 1997]); e *pull*, estratégia baseada no envio periódico de mensagens de solicitação de estado (*Are_you_alive?*) com consequente confirmação (*Yes_I_am!*). Em redes móveis sem fio, incluindo MANETs, a mais tradicional é a estratégia baseada em fofoca [Gracioli e Nunes 2007][Friedman *et al*. 2007], a qual é baseada no envio periódico de mensagens *gossip* que são repassadas de vizinhos locais para vizinhos remotos. Normalmente uma mensagem *gossip* carrega uma lista de identificadores e contadores de *heartbeat*.

Em MANETs, independente do uso de um algoritmo epidêmico, que dissemina a informação de nó para nó através de uma estratégia eficiente e escalável baseada em fofoca ou boatos, é preciso distinguir um comportamento defeituoso de uma movimentação do nó na rede.

3. Definições e Conceitos

Esta seção apresenta um conjunto de definições e conceitos relacionadas ao algoritmo de detecção de defeitos proposto na Seção 4, visando facilitar sua compreensão.

Definição 1. ***Alcance de transmissão.*** O alcance de transmissão (*range*) é o limite máximo de distância que uma mensagem emitida por um nó p_i pode chegar e é igual para todo $p_i \in \prod$.

Definição 2. ***Grupos de comunicação.*** O sistema pode ser representado por vários grupos de comunicação GC_i, formados por nós p_i e seus vizinhos locais em um dado instante de tempo t. Um nó p_j é dito vizinho local de p_i, se p_j pertence ao alcance de transmissão de p_i. Uma mensagem m enviada pelo nó p_i é recebida por todos os vizinhos corretos (não falhos) de p_i pertencentes a GC_i.

Das definições 1 e 2 tem-se que todos os nós do sistema têm o mesmo *range*, logo:

$\forall \{p_i, p_j\} \in \prod$: se $p_i \subset range_j$ então $p_j \subset range_i$

$\forall p_i \in \prod$: se $p_j \in GC_i$ então $p_j \subset range_i$

Definição 3. ***Regiões de cobertura.*** Assumindo que o alcance de transmissão de uma mensagem numa MANET é proporcional à potência do sinal de transmissão, a área de cobertura de um nó p_i (área dentro do alcance de transmissão) pode ser dividida em regiões de cobertura (vide Figura 1), onde cada região corresponde a um nível de potência do sinal de recepção (quanto mais longe do nó menor a potência).

Das definições 1 e 3, para uma dada região k, tem-se que

$\forall \{p_i, p_j\} \in \prod$: se $p_i \subset região_j^k$ então $p_j \subset região_i^k$.

A Figura 1 ilustra o *range* de um nó p_i e suas respectivas regiões de cobertura, as quais são distinguidas pela potência do sinal de recepção. Desta forma, através da análise da potência de recepção de cada mensagem recebida, p_i pode então localizar um nó p_j dentro de uma de suas regiões de cobertura.

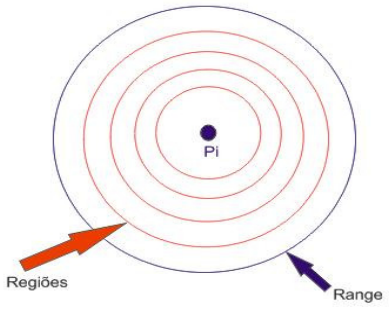
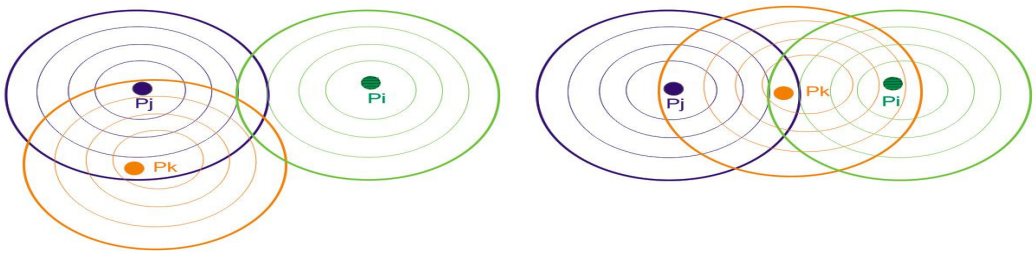


Figura 1. Regiões de Cobertura

Cada nó p_i pode se mover arbitrariamente na rede, mudando a configuração de seu grupo de comunicação GC_i. Desta forma, um nó p_i pode sair de um GC_k de p_k e entrar no GC_j de um nó p_j. Também pode ocorrer sobreposição de grupos de comunicação: um nó p_i pode pertencer aos grupos de comunicação dos nós p_j e p_k simultaneamente. A Figura 2 ilustra dois cenários com diferentes grupos de comunicação. Na Figura 2(*a*) p_k pertence a GC_j e não pertence ao GC_i. Na Figura 2(*b*) p_k pertence ao GC_j e ao GC_i. Em ambos os casos p_k também pertence ao GC_k.



(a) (b)

Figura 2. Cenários de Grupos de Comunicação

Definição 4. *Histórico de movimentação*. Cada nó p_i mantém um histórico H_i^j para todo p_j pertencente a GC_i, no qual são mantidas as informações sobre a região que cada nó vizinho p_j está. O histórico é atualizado em um intervalo de tempo finito e a cada atualização a informação mais atual sobre a região de p_j prevalece.

Definição 5. *Lista de suspeitos.* Cada nó p_i mantém uma lista de suspeitos SL_i. Um nó p_j é adicionado a SL_i se e somente se p_i não detecta movimentação de p_j nem recebe nenhuma mensagem de p_j dentro de um intervalo de tempo finito.

Definição 6. *Lista de mobilidade*. Cada nó p_i mantém uma lista de mobilidade ML_i. Um nó p_j é adicionado a ML_i se e somente se p_i detecta movimentação de p_j, a qual é baseada na análise das regiões de cobertura observadas a cada mensagem recebida.

Definição 7. *Lista de vizinhos.* Cada nó p_i mantém uma lista de vizinhos NBL_i, que corresponde a um conjunto de nós sobre os quais p_i recebe mensagens ou informações e acredita estarem corretos. Num dado instante de tempo a lista de vizinhos NBL_i pode conter: nós vizinhos locais (pertencentes ao GC_i); nós que se deslocaram para outra vizinhança (não mais pertencentes ao GC_i); e nós que nunca pertenceram ao GC_i.

4. Detector de defeitos com Suporte à Mobilidade

Esta seção apresenta a proposta de detector de defeitos não confiável para MANETs baseado na potência do sinal de recepção das mensagens. A Seção 4.1 detalha a adaptação do mecanismo básico de detecção do algoritmo Gossip, a Seção 4.2 detalha as estruturas de dados utilizadas e a Seção 4.3 detalha o algoritmo proposto. Finalmente a Seção 4.4 apresenta as propriedades e a prova de correção do algoritmo.

4.1 Adaptação do mecanismo básico para detecção de defeitos do algoritmo Gossip

Para construção da lógica de detecção de defeitos assume-se que todo nó monitora todos os outros nós do sistema através do envio de mensagens *gossip* de acordo com o protocolo *Gossip* básico [Renesse *et al.* 1998]. Porém, o mecanismo de detecção de defeitos foi adaptado como segue.

No *Gossip* básico a detecção de defeitos segue o seguinte algoritmo: a cada intervalo *Tgossip*, um nó escolhe aleatoriamente um ou mais vizinhos para enviar a mensagem *gossip*; no recebimento da mensagem, a lista de identificadores é unida com a lista que o receptor possui e o maior valor do contador de *heartbeats* de cada nó é mantido na lista final; cada nó mantém o instante do último incremento do contador de *heartbeat*; se o contador não for incrementado em um intervalo de *Tfail* unidades de

tempo então o nó é considerado suspeito, mas colocado na lista de suspeitos apenas após *Tcleanup* unidades de tempo; o *Tfail* é escolhido de acordo com a probabilidade de ocorrer uma detecção de defeito errônea (*Pmistake*) e o *Tcleanup* é escolhido de modo a fazer com que a probabilidade de receber uma mensagem *gossip* que evite uma suspeita errônea seja menor que *Pcleanup*.

É possível simplificar o protocolo *Gossip* permitindo que nós sejam removidos da lista depois de *Tcleanup* sem a necessidade de verificação do *Tfail* [Burns et al. 1999]. O ponto chave para esta simplificação é que o *Tcleanup* pode ser definido como múltiplo do *Tgossip* e precisa corresponder ao tempo necessário para as informações alcançarem outros nós dentro do tempo limite *Tcleanup* [Subramaniyan et al. 2005].

Neste trabalho, simplificamos as escolhas dos parâmetros do algoritmo *Gossip* fazendo *Tcleanup* ser ajustado para $Cfail \times Tgossip$, sendo *Cfail* o número de intervalos *Tgossip* que devem ser aguardados até que se confirme a suspeita de falha em um dado nó *p*. Como *Cfail* numa MANET depende do padrão de movimentação de cada nó, existe um *Cfail* para cada nó $p \in \prod$.

4.2 Estruturas de dados para suporte à mobilidade

Cada nó p_i pertencente a $\prod$ mantêm uma lista de suspeitos SL_i (Definição 5), uma lista de mobilidade ML_i (Definição 6) e uma lista de vizinhos NBL_i (Definição 7). O nó p_i mantém as informações dos seus vizinhos em uma estrutura de dados NB_i, onde para cada vizinho p_j é mantido:

- H_i^j - histórico de movimentação do nó p_j;
- TS_j - *timestamp* da última informação recebida do nó p_j;
- $Hcount_j$ - contador de *heartbeats* referente às mensagens *gossip* do nó p_j;
- $Cfail_j$ - contador de intervalos *Tgossip* que devem ser aguardados para sobrepor o tempo de desconexão local do nó p_j dado seu padrão de movimentação (trajetória de movimentação); e
- $Rpot_j$ - informação sobre a região (localização) de p_j na região de cobertura de p_i.

4.3 Algoritmo de detecção de defeitos com suporte à mobilidade

Para detalhar o funcionamento do algoritmo proposto esta subseção está dividida de acordo com as tarefas concorrentes do algoritmo, as quais se dividem em: 1 - envio de mensagens; 2 - recebimento de mensagens; e 3 - verificação de estado. O algoritmo completo é apresentado na Figura 3. No início, as estruturas de dados são ajustadas para que um nó p_i suspeite de todos os outros nós, exceto de si próprio.

Tarefa 1: envio de mensagens

Todo nó que executa o algoritmo de detecção de defeitos com suporte à mobilidade envia uma mensagem *gossip* por *broadcast* de rádio a cada intervalo *Tgossip*. Esta mensagem contém a sua lista de vizinhos (NBL_i) e a sua lista de contadores de *heartbeats* (*Hcount*). Este mecanismo faz com que a cada intervalo *Tgossip* um nó p_j dentro de uma das regiões de cobertura do nó p_i receba uma mensagem de p_i com informações sobre o nó p_i e seus vizinhos não suspeitos

(pertencentes ou não ao grupo de cobertura de p_i). Esta tarefa corresponde às linhas 7 a 9 do algoritmo de detecção com suporte à mobilidade ilustrado na Figura 3.

Note que o envio de NBL_i permite que a informação sobre um nó p_k vizinho de p_i e fora da vizinhança local de p_j, ou seja, fora da alcance de transmissão de p_j, possa ser difundida para p_j. Deste modo o nó p_j poderá passar a reconhecer o nó p_k como um de seus vizinhos (incluindo p_k na sua lista de vizinhos).

Tarefa 2: recebimento de mensagens

A cada mensagem *gossip* recebida de um nó p_j , o nó p_i verifica a potência do sinal de recepção referente a mensagem e atualiza $Rpot_j$ (linha 13). A região $Rpot_j$ onde p_j se localiza é então inserida no histórico H_i^j (linha 14). Se duas ou mais mensagens *gossip* forem recebidas de p_j num mesmo intervalo de verificação *Tcleanup*, apenas a informação mais recente será mantida no histórico.

Se p_j pertence a ML_i, então o nó p_j é adicionado em NBL_i e removido da lista ML_i (linhas 15 a 17). Por outro lado, se o nó p_j pertence a SL_i, é verificado quantos *Tgossip* o nó ficou como suspeito e $Cfail_j$ é atualizado (linha 20). Este procedimento permite que o nó p_i mantenha informação sobre o padrão de mobilidade do nó p_j e, num tempo finito, consiga evitar que ele seja novamente adicionado de maneira errada como suspeito. Note que p_j pode estar fora do alcance de transmissão mas não estar defeituoso. O nó é então adicionado novamente à lista NBL_i e removido da lista SL_i (linhas 21 e 22).

O nó p_i também atualiza informações gerais referentes a todos os nós não suspeitos p_k pertencentes a NBL_j (linhas de 24 a 35). O contador de *heartbeats* é atualizado para o maior valor mantido por p_i e p_j, e, em caso de atualização, o *timestamp* também é atualizado. Se p_k pertence a ML_i ou a SL_i então o nó p_k é adicionado em NBL_i e removido da ML_i ou SL_i, respectivamente. Se o nó p_k pertence a SL_i, o número de intervalos *Tgossip* que p_i ficou sem receber mensagens de p_k é computado e esse valor é atribuído a $Cfail_k$. Deste modo, um nó fora de GC_i pode se manter como não suspeito (em NBL_i).

Tarefa 3: verificação de estado

O mecanismo de verificação de estado serve para atualizar a visão local de um nó p_i sobre sua vizinhança ($NBL_i \cup ML_i$), incluindo os nós considerados móveis. Sua tarefa está dividida em duas fases: quando $p_j \in NBL_i$ e quando $p_j \in ML_i$.

Quando p_j pertence a lista de vizinhos NBL_i (linhas 39 a 48) o algoritmo verifica se o nó p_i recebeu informação dos nós vizinhos em um intervalo de tempo esperado ($Cfail_j$ * *TGossip*). Se o tempo de espera por uma mensagem do nó p_j não for superior ao esperado, o histórico de p_j é atualizado com a região referente ao recebimento da última mensagem de p_j (linha 48). Se o tempo de espera limite foi ultrapassado (teste na linha 40) o nó p_i atualiza o histórico H_i^j com uma informação vazia [-] (linha 41), indicando que p_i não recebeu mensagem ou informação de p_j no intervalo esperado. Além disto, quando extrapolado o tempo limite, se H_i^j [t-1] for igual ao H_i^j [t-2] e o contador de intervalos $Cfail_j = 0$, tem-se uma situação em que p_j não teve movimentação prévia, indicando uma potencial falha do nó. Logo, o nó é imediatamente adicionado à lista de suspeitos SL_i e removido da lista de vizinhos NBL_i (linhas 43 e 44). Por outro

lado, o algoritmo detecta uma movimentação de uma região para outra no intervalo de tempo [t-1, t-2] se H_i^j [t-1] $\neq$ H_i^j [t-2]. Deste modo, o nó é adicionado na lista de mobilidade ML_i e removido da sua lista de vizinhos NBL_i (linhas 46 e 47). Note que os nós em ML_i continuam na vizinhança de p_i, mas em movimento.

```
1.    H ← 0 ; Cfail ← 0 ; Rpot ← 0 ;
2.    NBL ← { };              //Lista de vizinhos
3.    ML ← { };               //Lista de Mobilidade
4.    SL ← { p_i, ..., p_n };  //Lista de Suspeitos
5.
6.    Task 1: Difusão
7.    a cada Tgossip faça
8.        Hcount_i = Hcount_i +1;
9.        broadcast (NBL_i, Hcount);
10.
11.   Task 2: Recebimento de Mensagem
12.   a cada mensagem gossip de p_j com (NBL_j, Hcount) faça
13.       atualize Rpot_j ;      //Rpot é a região correspondente a potência do sinal da msg
14.       atualiza H_i^j com Rpot_j;
15.       Se p_j ∈ ML_i então
16.               adiciona p_j em NBL_i;
17.               remove p_j de ML_i;
18.
19.       Se p_j ∈ SL_i então
20.               Cfail_j = (TL_i - TS_j ) / Tgossip;  // TL_i = hora atual de p_i e TS_j = timestamp_j
21.               adiciona p_j em NBL_i;
22.               remove p_j de SL_i;
23.
24.       ∀ p_k ∈ NBL_j com i ≠ k faça
25.               atualiza Hcount_i^k com Max(Hcount_i^k, Hcount_j^k)
26.               Se Hcount_j^k > Hcount_i^k então TS_k = TL_i;
27.
28.               Se p_k ∈ ML_i então faça
29.                       adiciona p_k em NBL_i;
30.                       remove p_k de ML_i;
31.
32.               Se p_k ∈ SL_i então faça
33.                       Cfail_k = (TL_i - TS_k ) / Tgossip;
34.                       adiciona p_k em NBL_i;
35.                       remove p_k de SL_i;
36.
37.   Task 3: Verificação de Estado
38.   a cada TCleanup faça
39.       ∀ p_j ∈ NBL_i com i ≠ j faça
40.               se TL_i - TS_j > Cfail_j * TGossip então
41.                       atualiza H_i^j com '-';
42.                       se H_i^j [t-1] = H_i^j [t-2] e Cfail_j = 0 então
43.                               adiciona p_j em SL_i;
44.                               remove p_j de NBL_i;
45.                       senão
46.                               adiciona p_j em ML_i;
47.                               remove p_j de NBL_i;
48.               senão atualize H_i^j (Rpot_j);
49.       ∀ p_j ∈ ML_i com i ≠ j faça
50.               se TL_i - TS_j > Tgossip + (Cfail_j * Tgossip) então
51.                       adiciona p_j em SL_i;
52.                       remove p_j de ML_i;
```

Figura 3. Algoritmo de detecção de defeitos com suporte à mobilidade

Quando p_j pertence a ML_i (linhas 49-52) é verificado se a última movimentação ocorreu a mais do que $Tgossip$ + ($Cfail_j$ * $Tgossip$) unidades de tempo. Caso não, o período de movimentação previamente conhecido ainda não expirou, indicando uma possível movimentação em curso, logo nada é feito e aguarda-se a próxima verificação

de estado. Caso sim, o nó é considerado suspeito e retirado da lista de mobilidade. A espera por um número de intervalos múltiplo de $Cfail_j$ procura computar o tempo de movimentação antes de adicionar o nó como suspeito. Isto faz com que um nó que possui um padrão de mobilidade não seja suspeito por apenas sair do alcance de transmissão.

4.4 Propriedades e Prova de Correção do Algoritmo

O algoritmo de detecção de defeitos com suporte à mobilidade apresentado na Seção 4.3 atende as propriedades de completude forte e exatidão fraca após um tempo definidas em [Chandra e Toueg 1996], ou seja, é um algoritmo da classe ◊S. Tais propriedades asseguram que, após um tempo, os nós defeituosos serão suspeitos pelos nós corretos (completude forte) e que existirá também um instante de tempo t após o qual um nó correto não será considerado suspeito por nenhum outro nó (exatidão fraca após um tempo).

A seguir, apresenta-se um argumento de prova para o algoritmo proposto, considerando as propriedades de um detector de defeitos da classe ◊S.

Considere um nó correto p_i e um nó falho p_f . Para que o algoritmo possua a propriedade de *completude forte*, deve-se demonstrar que eventualmente um nó $p_f \in \prod$ será incluído na lista de suspeitos SL_i de todo nó $p_i \in \prod$, sendo $i \neq f$. Esta condição segue das seguintes verificações:

(1) Todo nó p_f pertencente a $\prod$ está inicialmente em SL_i. Para fazer parte de NBL_i o nó p_f deve ter enviado pelo menos uma mensagem *gossip* em um intervalo *Tgossip*.

(2) Um nó p_f não enviará mais mensagens *gossip*. Todo nó p_i espera $Cfail_f$ intervalos antes de adicionar p_f como falho. Logo, existirá um instante de tempo após o qual p_f será adicionado como suspeito em SL_i de todo nó $p_i \in \prod$.

De maneira similar, para demonstração da propriedade de *exatidão fraca após um tempo*, deve-se verificar que existe um instante de tempo t a partir do qual um nó correto p_i não vai ser suspeito por nenhum outro nó correto do sistema, ou seja, após o instante t o nó p_i não vai fazer parte de nenhum conjunto SL_j. Esta condição é verificada como segue.

(1) Se p_i pertence ao GC_j, após um intervalo *Tgossip*, p_i enviará uma mensagem *gossip* para p_j, que adicionará o nó p_i em sua lista de vizinhos NBL_j. Essa informação será difundida para todos vizinhos p_k do nó p_j, onde $p_k \in GC_j$. Deste modo, todo nó p_i que se movimenta numa trajetória desconhecida, mas que obedece a um padrão de movimentação no qual após um tempo sempre passa na vizinhança de algum nó correto, terá sua informação de sobrevivência difundida para todos os nós corretos. Todo nó correto irá receber informações sobre o nó p_i e remover p_i da lista de suspeitos, adicionando p_i em sua lista de vizinhos NBL_k.

(2) Se o nó p_i estiver na lista de suspeitos de algum nó p_j pertencente a $\prod$, o nó p_i terá a sua informação de $Cfail_i$ atualizada, fazendo com que o nó p_i possa ter o mesmo intervalo de movimentação dentro de sua trajetória sem ser adicionado novamente como suspeito.

5. Testes e Avaliações, Simulações e Comparações

Nesta seção é apresentado o resultado das simulações feitas com o detector de defeitos baseado na potência do sinal (DD). Nos experimentos o modelo de simulação foi configurado para operar num campo de $300m^2$, com uma frequência de transmissão de 2.412 ~ 2.462 GHz (IEEE 802.11g) e uma largura de banda de 54 Mbps. O alcance de transmissão de cada nó foi fixado em 50 metros. Para fins de comparação, os parâmetros de simulação foram os mesmos utilizados por Friedman e Tcharny (2005): número de nós, velocidade de movimentação, atraso de comunicação e modelo de movimentação (*Random Waypoint*). Nas simulações foram utilizados diferentes valores para as velocidades máximas de movimentação (de 2 a 8 m/s), número de nós (de 30 a 60 nós).

O primeiro experimento visou identificar o comportamento do número de falsas suspeitas e do tempo de detecção (T_D) frente à variação da quantidade de nós e da velocidade de movimentação. A Figura 4(a) ilustra que o aumento da quantidade de nós e da velocidade dos nós gera um aumento proporcional no número de falsas suspeitas. A proporcionalidade deriva da propriedade linear do algoritmo na manipulação das regiões. A Figura 4(b) ilustra que o T_D decresce rapidamente com o aumento da velocidade de movimentação. Isto deve-se ao fato de que quando se aumenta a velocidade de movimentação dos nós, o parâmetro *CFail* do algoritmos (*timeout*) tende a diminuir, pois se um nó faz uma dada rota, esta será percorrida num menor tempo, diminuindo assim o *CFail*, pois o detector manipula o contador *CFail* com intervalos de espera até classificar um nó como falho.

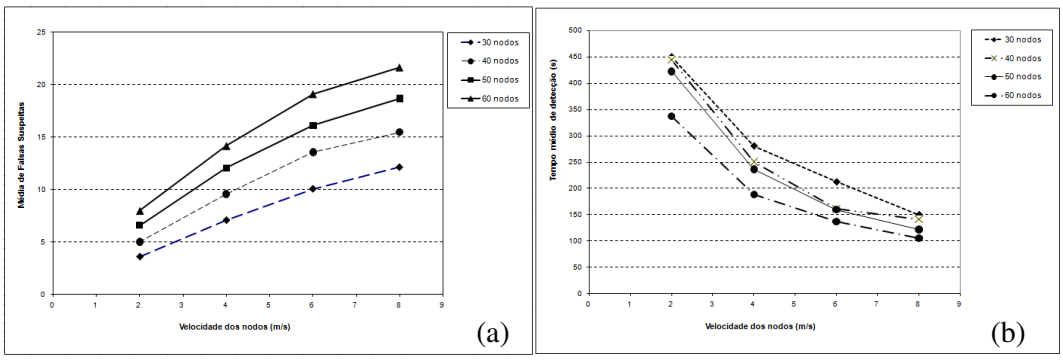


Figura 4. Comportamento do (a) número de falsas suspeitas e (b) tempo médio de detecção

O segundo experimento analisa o comportamento do T_D e da variabilidade do número de falsas suspeitas (Figura 5(a)) e do T_D mínimo frente a variabilidade da velocidade dos nós (Figura 5(b)). Da Figura 5(a), observa-se que o decremento do T_D é proporcional ao aumento do número de falsas suspeitas, o que era esperado. Isso ocorre porque o nó varre sua lista local antes de cada emissão de *broadcast* para enviar sempre sua lista mais atual e, deste modo, tende a suspeitar mais vezes dos nós vizinhos que não enviaram mensagens no último intervalo *CFail*. Quando há menos nós na rede, ocorre menos troca de mensagens (todos nós pertencentes a um dado CG observam as mensagens de *broadcast*), o que tende a aumentar *CFail*. A Figura 5(b) ilustra um tempo de detecção mínimo baixo e relativamente independente do aumento da velocidade dos nós. Isto deve-se ao fato de que quando uma falha ocorre dentro do alcance de transmissão a detecção é rápida, se comparado com o tempo de detecção

quando um nó está fora do alcance de transmissão. É importante observar que este comportamento corresponde a um monitoramento de um nó que se encontra dentro do alcance de transmissão em mais de uma rodada, ou seja, similar ao de uma rede fixa.

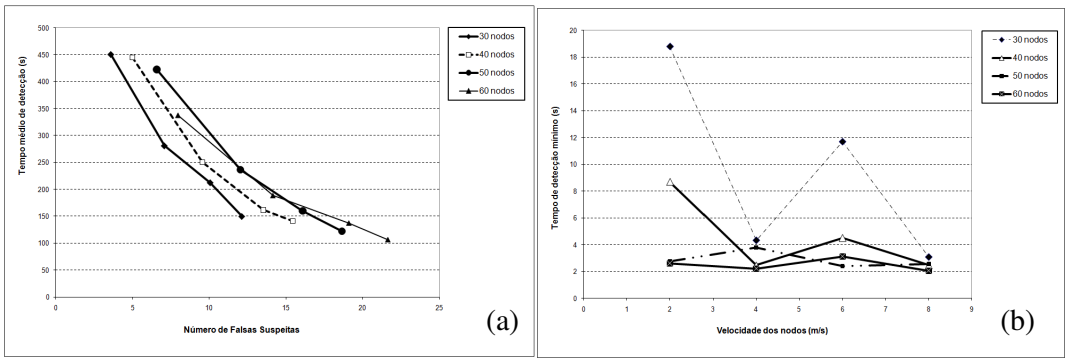


Figura 5. Comportamento (a) do T_D frente a variabilidade do número de falsas suspeitas e (b) do T_D mínimo frente a variabilidade da velocidade dos nós

O terceiro experimento realizou a comparação com outros detectores de defeitos, a fim de identificar o comportamento do número de falsas suspeitas e do tempo de detecção (T_D), confrontando os resultados obtidos pelo detector DD contra os resultados gerados pelos detectores propostos em [Friedman e Tcharny 2005][Sridhar 2006][Sens *et al*. 2008]. Os testes realizados utilizaram uma configuração de rede com 30 nós movendo-se a uma velocidade de 2 m/s. O gráfico da Figura 6(a) demonstra que o DD gerou menos falsas suspeitas em comparação aos outros detectores de defeitos, ficando com um valor muito próximo aos detectores propostos por Sridhar e Friedman. O detector proposto por Pierre Sens corrige este elevado número de falsas suspeitas após suas mensagens de *Query* serem disseminadas na rede, porém o detector comete um elevado número de falsas suspeitas se considerado o tempo total de simulação. O baixo número de falsas suspeitas do DD deve-se ao uso do histórico de movimentação e à utilização do contador *CFail*. No que se refere ao tempo de detecção T_D, a Figura 6(b) ilustra que o detector de Pierre Sens é o que resulta no melhor desempenho de detecção e que o DD consegue resultado melhor que os detectores de Friedman e Sridhar. Este resultado demonstra que o DD é competitivo em relação ao desempenho. Juntos, os resultados em relação ao tempo para detecção de falhas (T_D) e a quantidade de falsas suspeitas, permitem observar que o DD apresenta uma boa qualidade de serviço em relação aos outros detectores similares.

A qualidade do detector de defeitos baseado na potência do sinal (DD) deriva da eliminação de suspeita imediata de um nó p_j que sai do alcance de transmissão de um nó p_i para percorrer uma dada rota de movimentação. O nó percebe uma movimentação de um vizinho que está saindo do seu alcance de transmissão e ajusta o parâmetro *CFail* para aguardar um intervalo de tempo adicional antes de colocar o nó na lista de suspeitos.

Salienta-se que a percepção de movimentação foi realizada a partir de um histórico de regiões que considera as movimentações nos últimos dois intervalos de tempo de transmissão. O intervalo pode ser suficiente caso o nó repita a sua rota num intervalo menor do que duas vezes o período de atualização do histórico, mas deve ser

revisto e é adaptável conforme as novas rotas descobertas (que levam mais tempo). Para adaptar os intervalos de tempo, o DD não precisa conhecer quantos *hops* um nó está do outro, ou informações sobre o *jitter* da rede. Ao invés disto o algoritmo consegue adaptar e obter essas informações apenas observando o funcionamento da rede através da troca de mensagens.

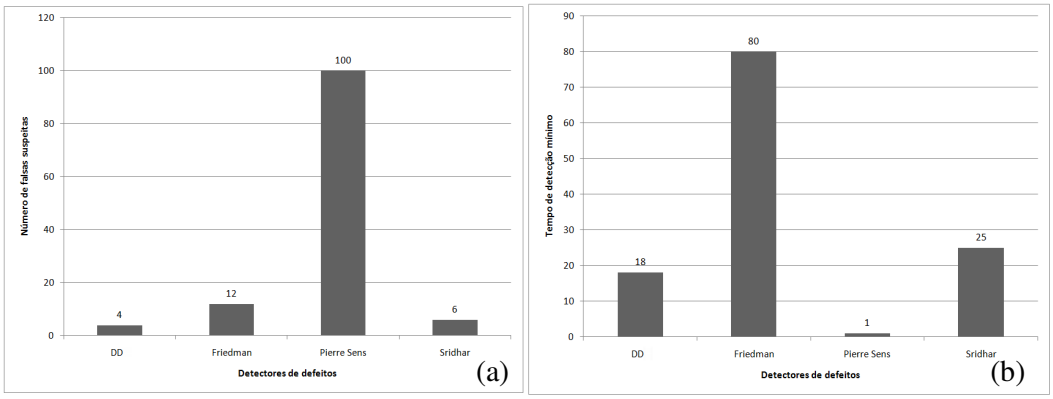


Figura 6. Qualidade do DD frente a outros detectores (a) quanto ao número de falsas suspeitas e (b) quanto ao tempo de detecção (T_D mínimo)

6. Trabalhos Relacionados

Na literatura de detecção de defeitos a mobilidade dos nós foi tratada por diversos autores. Friedman e Tcharny (2005) procuraram representar a mobilidade permitindo que o *Tcleanup* do algoritmo *Gossip* [Renesse *et al.* 1998] pudesse ser incrementado com múltiplos do atraso de rede a um *hop*, suportando assim lacunas no recebimento de mensagens de *heartbeat*. Hutle (2004) também propôs um algoritmo derivado do *Gossip* mas a intenção foi garantir QoS através da inserção de uma variação conhecida do atraso de comunicação (sincronismo fraco). Zia *et al.* (2009) procuraram detectar mobilidade no detector de defeitos usando um *round* adicional de difusão de mensagens e permitindo que os nós que se moveram para fora do alcance de transmissão sejam marcados como suspeitos até que alguma contra-resposta seja obtida. Porém nestas técnicas, a taxa média de erros pode ser alta se os nós se movem frequentemente para fora da região de cobertura.

Alternativamente, Sridhar (2006) propôs um detector de defeitos que trata da mobilidade dos nós mantendo uma lista de nós suspeitos de falha e difundindo essa lista para descobrir se nós marcados como suspeitos não estão em uma outra vizinhança. Além da lista, são usadas duas camadas: uma camada de detecção local (responsável por gerar a lista de vizinhos) e uma camada de detecção de mobilidade. Sens *et al.* (2008) não utilizam fofoca e exploraram a técnica de detecção de defeitos *query-response* para chegar a um detector de defeitos que não depende do monitoramento de tempo.

Diferente de tudo que se encontrou na literatura, este trabalho explorou informações de potência do sinal de recepção que permitem identificar uma

movimentação dos nós vizinhos que estão dentro do alcance de transmissão, habilitando a distinção entre nó em mobilidade e nó defeituoso.

7. Conclusões e Trabalhos Futuros

Detecção de defeitos ainda tem sido alvo de pesquisas para sua utilização eficiente em redes móveis, onde o dinamismo da rede, a escassez de recursos, a comunicação sem fio por difusão de mensagens e a dificuldade de se diferenciar entre um nó defeituoso e um nó móvel são fatores importantes. Este trabalho apresentou uma nova abordagem para implementação de detectores de defeitos não confiável com suporte à mobilidade de nós de redes dinâmicas: distinguir um nó defeituoso de uma mobilidade através da análise da potência do sinal de recepção das mensagens que chegam a um dado nó.

O algoritmo proposto demonstrou ser eficaz quanto a velocidade de detecção de defeitos, quando estes ocorrem em instantes em que o nó defeituoso está dentro do alcance de transmissão do nó monitor. Esta situação é equivalente a uma situação sem mobilidade. Quando a falha ocorre fora do alcance de transmissão de um determinado nó, o tempo máximo de detecção da falha do nó será proporcional ao período de tempo de comunicação (*TGossip*) e será determinado pelo fator *CFail*. Este comportamento é similar aos demais algoritmos que adaptam seu *Timeout*. Por outro lado, em situações de mobilidade que a rota de movimentação segue um padrão de movimentação conhecido (repetição de rota), o algoritmo adapta-se as rotas de cada nó rapidamente e evita falsas suspeitas quando um nó correto sai do alcance de transmissão apenas para fazer a sua rota. Neste cenário o resultado observado foi um baixo número de falsas detecções quando comparado com algoritmos similares.

Em trabalhos futuros vislumbra-se explorar: (1) a identificação de mobilidade no próprio nó, o que pode permitir que ele informe outros nós sobre sua movimentação ou mesmo permitir a trocar de comportamento do detector; (2) o impacto do refinamento do tamanho das regiões consideradas e do tamanho do histórico, visando instrumentalizar o detector com uma noção mais precisa da rota de movimentação.

Referências

Aguilera, M. K.; Chen, W.; Toueg, S. (1997) “Heartbeat: A timeout-free failure detector for quiescent reliable communication”. Workshop on Distributed Algorithms, pp. 126–140.

Aguilera, M. K.; Chen, W.; Toueg, S. (1998) “Failure Detection and Consensus in the Crash-Recovery Model”. Technical Report 98-1676. Department of Computer Science, Cornell University, v. 1499, pp.231-245.

Burns, M.; George, A; Wallace, B. (1999) “Simulative Performance Analysis of Gossip Failure Detection for Scalable Distributed Systems”. *Cluster Computing*. 2(3):207-217.

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

Chen, W; Toueg, S.; Aguilera, M. K. (2002) “On the quality of service of failure detectors”, *IEEE Transactions on Computing*, 51(1):13-32.

Felber, P.; Défago, X.; Guerraoui, R.; Oser, P. (1999) “Failure detectors as first class objects”. In: International Symposium on Distributed Objects and Applications (DOA’99). Edinburgh, Scotland. pp.132–141. IEEE Computer Society.

Friedman, R.; Gavidia, D.; Rodrigues, L.; Viana, A. C.; Voulgaris, S. (2007) “Gossiping on MANETs: the Beauty and the Beast”, *ACM Operating Systems Review*, pp.67-74.

Friedman, R.; Tcharny, G. (2005), "Evaluating failure detection in mobile ad-hoc networks", *International Journal of Wireless and Mobile Computing*, 1(8):23.

Fischer, M. J.; Lynch, N. A.; Paterson, M. S. (1985) “Impossibility of distributed consensus with one faulty process”. *Journal of the ACM*, v.32, n.2, pp. 374-382.

Gomez, J.; Campbell, A. T. (2007) “Using Variable-Range Transmission Power Control in Wireless Ad Hoc Networks”, *IEEE Trans on Mobile Computing*, 6(1), pp.87-99.

Gracioli, G.; Nunes, R. C. (2007) “Detecção de defeitos em redes móveis sem fio: uma avaliação entre as estratégias e seus algoritmos”. In: Anais do Workshop de Testes e Tolerância a Falhas (SBRC/WTF), Belém/PA, pp.159-172.

Hutle, M. (2004) “An efficient failure detector for sparsely connected networks”. In: Proceedings of the International Conference on Parallel and Distributed Computing and Networks. Innsbruck: Austria, pp.369-374.

Loo, J.; Mauri, J. L.; Ortiz, J. H. (2012) “Mobile Ad Hoc Networks: current status and futures trends”. CRC Press, 538p.

Mahapatro, A.; Khilar, P.M., (2013) “Fault Diagnosis in Wireless Sensor Networks: A Survey,” *Communications Surveys & Tutorials*, IEEE , vol.15, no.4, pp.2000-2026.

Renesse, R.; Minsky, Y.; Hayden, M. (1998) “A Gossip-style failure detection service”. In: Proceedings of the IFIP Int. Conf. on Distributed Systems and Platforms and Open Distributed Processing (Middleware’ 98), pp.55-70.

Sens, P., Greve, F., Arantes, L., Bouillaguet, M., Simon. V. (2008) “Um detector de falhas assíncrono para redes móveis e auto-organizáveis”. In: Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos – SBRC’ 2008, pp.931-944.

Sridhar, N. (2006) “Decentralized local failure detection in dynamic distributed systems”. In: IEEE Symp on Reliable Distributed Systems (SRDS'06). pp.143-154.

Subramaniyan, R.; Raman, P.; George, A. D.; Radlinski, M. (2006) “GEMS: Gossip-Enabled Monitoring Service for Scalable Heterogeneous Distributed Systems”. *Cluster Computing*, 9(1), pp.101-120.

Zia, H. A.; Sridhar, N; Sastry, S. (2009) “Failure detectors for wireless sensor-actuator systems”. *Ad Hoc Networks*, 7:1001-1013.

Proposta de Implementação de Memória Compartilhada Dinâmica Tolerante a Falhas

Mateus Braga[1] **e Eduardo Alchieri**[1]

[1]Departamento de Ciência de Computação
Universidade de Brasília
Brasília - Brasil

Abstract. *Quorum systems are useful tools for implementing consistent and available storage in the presence of failures. These systems usually comprise a static set of servers that provide a fault-tolerant read/write register accessed by a set of clients. This paper proposes an implementation for* FREESTORE*, a set of fault-tolerant protocols that emulates a register in dynamic systems in which processes are able to join/leave the servers set by reconfigurations. A set of experiments analyses the performance of the implementation and brings some light to the shared memory reconfiguration procedure.*

Resumo. *Sistemas de quóruns são úteis na implementação consistente e confiável de sistemas de armazenamento de dados em presença de falhas. Estes sistemas geralmente compreendem um conjunto estático de servidores que implementam um registrador acessado através de operações de leitura e escrita. Este artigo propõe uma implementação para o* FREESTORE*, um conjunto de protocolos tolerantes a falhas capazes de emular um registrador em sistemas dinâmicos, onde processos podem entrar e sair, durante sua execução, através de reconfigurações. Um conjunto detalhado de experimentos avalia o desempenho dos protocolos implementados e possibilita uma maior compreensão a respeito do processo de reconfiguração de memória compartilhada.*

1. Introdução

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

Sistemas de quóruns foram inicialmente estudados em ambientes estáticos [Gifford 1979, Attiya et al. 1995, Malkhi and Reiter 1998], onde não é permitida a entrada e saída de servidores durante a execução do sistema. Mais recentemente, surgiram também propostas para sistemas de quóruns reconfiguráveis [Alchieri et al. 2014, Aguilera et al. 2011, Gilbert et al. 2010, Martin and Alvisi 2004], i.e., sistemas em que os protocolos permitem modificações no conjunto de servidores que implementam a memória compartilhada (registrador). O processo de modificar o conjunto de processos que compõem o sistema chama-se *reconfiguração* e faz com que o sistema se mova de uma configuração (visão) antiga para uma nova configuração atualizada.

Estas propostas para sistemas reconfiguráveis podem ser dividas em soluções que empregam consenso [Lamport 1998] na definição da nova visão a ser instalada no sistema de forma que todos os processos concordem com a nova visão [Alchieri et al. 2014,

Gilbert et al. 2010, Martin and Alvisi 2004] e soluções que não precisam desta primitiva de sincronização [Alchieri et al. 2014, Aguilera et al. 2011]. Dentre estas soluções destaca-se o FREESTORE [Alchieri et al. 2014] por ser um conjunto de protocolos modulares que implementam reconfigurações tanto usando consenso quanto sem usar esta primitiva. De fato, através da modularidade do FREESTORE é possível aglutinar todos os requisitos de sincronia necessários para a reconfiguração do sistema em um módulo específico, o qual pode ser implementado destas duas formas. Outra característica importante desta solução é que os protocolos de leitura e escrita (R/W) são desacoplados dos protocolos de reconfiguração, simplificando a adaptação de protocolos estáticos de R/W para ambientes dinâmicos.

Mesmo com todas estas propostas para um sistema de quóruns reconfigurável, ainda falta uma implementação para os mesmos que possa ser usada no desenvolvimento de aplicações distribuídas. Este trabalho visa preencher esta lacuna e apresenta uma implementação para os protocolos do FREESTORE. Este sistema foi escolhido por fornecer as duas formas de reconfiguração acima discutidas, possibilitando uma melhor análise e compreensão a respeito do processo de reconfiguração destes sistemas.

O restante deste artigo está organizado da seguinte maneira. A Seção 2 apresenta o modelo de sistema assumido para a implementação dos protocolos, enquanto que a Seção 3 apresenta uma visão geral sobre o FREESTORE. A Seção 4 apresenta a nossa proposta de implementação e a Seção 5 discute uma série de experimentos realizados com a implementação desenvolvida. Finalmente, a Seção 6 apresenta nossas conclusões.

2. Modelo de Sistema

Nossa implementação considera um sistema distribuído completamente conectado composto pelo conjunto universo de processos U, que é dividido em dois subconjuntos distintos: um conjunto infinito de servidores $\Pi = \{s_1, s_2, ...\}$; e um conjunto infinito de clientes $C = \{c_1, c_2, ...\}$. Clientes acessam o sistema de armazenamento implementado por um subconjunto dos servidores (uma *visão*) executando operações de leitura e escrita (Seção 4.1). Cada processo do sistema (cliente ou servidor) possui um identificador único (Seção 4.2) e está sujeito a falhas por parada (*crash*). A chegada dos processos no sistema segue o modelo de chegadas infinitas com concorrência desconhecida mas finita [Aguilera 2004]. Além disso, os processos estão conectados através de canais confiáveis, os quais são implementáveis através do uso do protocolo TCP/IP.

Os protocolos de reconfiguração do FREESTORE são implementáveis em um sistema distribuído assíncrono, onde não existem limites para o tempo de transmissão de mensagens ou processamentos locais nos processos[1]. Além disso, cada servidor do sistema tem acesso a um relógio local usado para iniciar reconfigurações (Seção 4.4).

Para cada visão v são permitidas no máximo f falhas de processos, sendo $f \leq \lfloor \frac{n-1}{2} \rfloor$ onde n é o número de processos presentes em v, com quóruns de tamanho $v.q = \lceil \frac{n+1}{2} \rceil$. Finalmente, um processo que deseja deixar o sistema deve aguardar até que uma visão sem sua presença seja instalada e o número de reconfigurações concorrentes com uma operação de R/W deve ser finito [Alchieri et al. 2014].

[1]Apesar de não ser a única alternativa, é possível utilizar um protocolo de consenso durante uma reconfiguração (Seção 3). Neste caso precisamos de um sistema parcialmente síncrono [Dwork et al. 1988].

3. Visão Geral do FREESTORE

O FREESTORE [Alchieri et al. 2014] compreende um conjunto de protocolos para a realização de reconfigurações em sistemas de quóruns. Sua organização é modular e apresenta uma desejável separação de conceitos diferenciando claramente os processos de gerar uma nova visão[2], de reconfiguração do sistema (i.e., instalar uma visão atualizada) e de leitura e escrita no registrador.

Este sistema introduz a ideia de geradores de sequências de visões, os quais englobam todos os requisitos de sincronia necessários para reconfigurar o sistema. O resultado da execução destes geradores são sequências de visões para serem instaladas e com isso reconfigurar o sistema. De acordo com o nível de sincronia que o ambiente de execução deve apresentar, um gerador de sequência de visões pode ser construído de pelo menos duas formas diferentes.

- *Geradores para sistemas parcialmente síncronos*: esta abordagem utiliza um protocolo de consenso de forma que todos os servidores envolvidos concordam (entram em acordo) com a nova sequência de visões a ser instalada. Na verdade todos os servidores geram uma única sequência com uma única visão para ser instalada no sistema. Protocolos de consenso [Lamport 1998, Lamport 2001, Castro and Liskov 2002] são bem conhecidos e utilizados mas possuem a limitação de não serem implementáveis em um sistema distribuído completamente assíncrono quando falhas são previstas [Fischer et al. 1985].
- *Geradores para sistemas assíncronos*: outra possível construção para os geradores de sequências de visões é sem o uso de consenso, não adicionando nenhum requisito de sincronia para sua execução. Como não temos acordo nesta abordagem, diferentes geradores podem gerar diferentes sequências mas através das propriedades de intersecção dos quóruns o FREESTORE garante que todos os servidores acabam convergindo e instalando a mesma sequência de visões.

Uma das motivações deste trabalho é a análise prática dos ganhos e das perdas relacionadas com a utilização de cada uma destas abordagens na implementação de geradores de sequência de visões.

Em termos gerais, uma reconfiguração do FREESTORE segue três etapas básicas:

Primeira etapa
O sistema recebe os pedidos de entrada (*join*) e saída (*leave*) de processos (servidores). Esses pedidos são acumulados por um período de tempo até que o sistema passe para a segunda etapa.

Segunda etapa
O gerador de sequência de visões escolhido é executado para geração de uma nova sequência de visões para atualizar o sistema. As visões desta sequência sempre são mais atuais do que a visão atual instalada no sistema, contemplando os pedidos de atualizações coletados na primeira etapa. O protocolo vai para a terceira etapa quando uma sequência é gerada.

Terceira etapa
Uma sequência de visões gerada é instalada de forma que, no final, uma única

[2]Basicamente, uma visão nada mais é do que um conjunto com os identificadores dos servidores presentes no sistema.

visão mais atual w é efetivamente instalada no sistema. Mesmo que processos diferentes gerem sequências diferentes (este cenário é possível quando o gerador não utiliza protocolos de consenso), os protocolos do FREESTORE garantem que todos os servidores acabam instalando w.

Como os protocolos de reconfiguração são desacoplados dos protocolos de leitura e escrita (R/W) no registrador, qualquer protocolo de R/W pode ser adaptado para ambientes dinâmicos através do FREESTORE. Neste trabalho consideramos o clássico ABD [Attiya et al. 1995], que implementa um sistema de quóruns com semântica atômica [Lamport 1986] das operações de leitura e escrita. Além disso, o FREESTORE garante que todos os pedidos de entrada ou saída são executados e nenhum processo entra ou é removido (sai) do sistema sem ter realizado tal pedido.

4. Proposta de implementação dos protocolos do FREESTORE

Esta seção apresenta a nossa proposta de implementação para os protocolos do FREESTORE, cujo sistema resultante representa uma memória compartilhada dinâmica. Primeiramente apresentaremos os módulos que compõem o sistema e sua API, para então discutir as implementações realizadas.

Nossas implementações foram realizadas na linguagem de programação *Go*[3], que é aberta, compilada e com coletor de lixo. Além disso, esta linguagem apresenta várias características de programação concorrente, como por exemplo as *goroutines*, que provêem uma abstração eficiente de *threads*.

4.1. Módulos do Sistema e APIs

A implementação proposta busca manter a modularidade do FREESTORE. Assim, o sistema pode ser dividido em módulos que interagem entre si conforme mostra a Figura 1. Esses módulos são: visão atual, registrador *R/W*, reconfiguração e gerador de sequências de visões. O registrador R/W precisa acessar a visão atual para saber se uma operação de R/W está utilizando a visão atual do sistema. O módulo de reconfiguração altera a visão atual e o valor armazenado no registrador ao final de cada reconfiguração e, além disso, executa o gerador de sequências de visões que, por sua vez, retorna uma nova sequência de visões quando gerada. Note que o gerador ainda possui um módulo de consenso interno que pode ser empregado quando necessário, conforme já discutido.

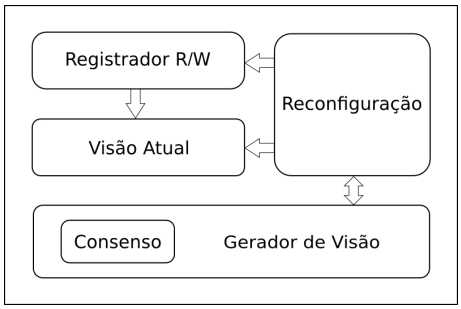


Figura 1. Módulos do sistema.

A implementação representa um sistema de quóruns com capacidade de reconfiguração e é dividida em clientes e servidores. Todas as funcionalidades do sistema

[3]Disponível em `http://golang.org`.

são implementadas em forma de bibliotecas, facilitando a integração do sistema dentro de uma aplicação maior que precise de memória compartilhada. As bibliotecas apresentam as seguintes APIs:

Funções do cliente:

Read() (int, error) – Retorna o valor armazenado no sistema e um erro, o valor só é válido se o erro for nulo.

Write(v int) error – Escreve o valor *v* no sistema e retorna um erro caso ocorra algum problema.

Funções dos servidores

Run(bindAddr string, initialView *view.View, useConsensus bool) – Inicializa a execução de um servidor do sistema.

bindAddr: o endereço em que o servidor irá fornecer seus serviços.

initialView: visão inicial do sistema. Se o servidor for membro dessa visão inicial, todos os seus serviços serão ativados no final da inicialização, caso contrário, o servidor irá enviar o pedido de *Join* para todos os servidores dessa visão.

useConsensus: se *true*, o servidor irá utilizar o gerador de sequência de visões com consenso, caso contrário não utilizará o protocolo do consenso. Esse parâmetro deve ser igual em todos os servidores em execução no sistema.

As próximas seções discutem os aspectos principais relacionados com a implementação das APIs acima descritas. Começamos apresentando as estruturas básicas que são vastamente utilizadas em nossa implementação, para então discutir a implementação dos protocolos de R/W e de reconfiguração do sistema.

4.2. Estruturas Básicas: Identificadores de Servidores, *Updates* e Visões

As estruturas abaixo servem como tipos básicos para a implementação dos protocolos.

Identificadores de Servidores: Um servidor é identificado pelo seu endereço.

```
1 type Process struct {
2   Addr string
3 }
```

***Updates*:** Representam atualizações no grupo dos participantes do sistemas. Um *update* é composto por um tipo (entrada/saída) e um servidor.

```
1 type Update struct {
2   Type  updateType
3   Process Process
4 }
5 type updateType string
6 const (
7   Join updateType = "+"
8   Leave updateType = "-"
9 )
```

Visão: Representa a composição do sistema, sendo formada por um conjunto de *updates*.

```
1 type View struct {
2   Entries map[Update]bool
3 }
```

Duas visões são iguais quando suas *Entries* são iguais. Dizemos que uma visão A é mais atualizada que uma visão B se A contém todas as atualizações que B

possui mais algumas outras atualizações. Um processo P pertence a uma visão caso a mesma possua o *update* $+P$ (*join*) e não possua o *update* $-P$ (*leave*). Cada servidor contém a visão atual do sistema, que é utilizada para verificar o quão atual são os eventos recebidos, sendo atualizada pelo módulo de reconfiguração.

4.3. Implementação do Protocolo de Leitura e Escrita

O protocolo de leitura e escrita em um registrador implementado é uma adaptação do clássico ABD [Attiya et al. 1995] para utilização com os protocolos do FREESTORE. A única mudança é que as requisições dos clientes agora precisam conter a visão mais atual conhecida pelo cliente. Um servidor que recebe um pedido com uma visão antiga apenas informa o cliente sobre a nova visão instalada, com a qual o mesmo deve reenviar seu pedido (reinicia a fase da operação que está executando).

Essa parte do sistema implementa a interface utilizada pelo cliente para processamento dos pedidos de *read* e *write*. O registrador é uma variável de tipo genérico, assim qualquer valor pode ser armazenado. Associado ao registrador temos um *mutex* para realizar o controle de concorrência entre *goroutines* concorrentes e para suspender as operações de R/W enquanto o servidor estiver atualizando a sua visão.

A Figura 2 ilustra o processamento das requisições em um servidor: ao receber uma dessas requisições, o servidor utiliza uma *goroutine* para atender a tal solicitação. Primeiro é verificado se a visão utilizada pelo cliente é atual (*cvMutex* controla a concorrência entre estes pedidos e reconfigurações que alteram a visão atual) para então acessar o valor do registrador (*registerMutex* controla a concorrência entre estes pedidos e também suspende a execução de operações de R/W enquanto a visão é atualizada).

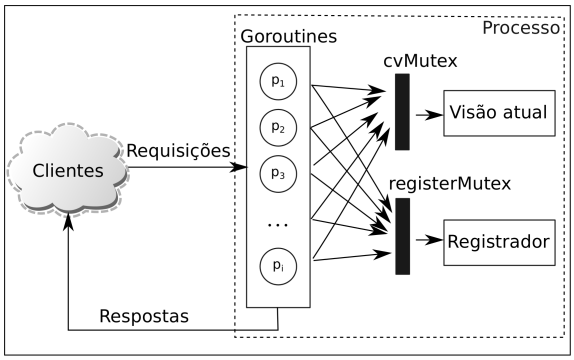


Figura 2. Processamento das requisições dos clientes.

As mensagens trocadas entre clientes e servidores contém a seguinte estrutura:

```
1 type Value struct {
2   Value   interface{} // Valor do registrador
3   Timestamp int // O timestamp do valor do registrador
4   View view.View // A visão do cliente
5   Err error   // Erro caso houver
6 }
```

Um cliente realiza as operações de leitura e escrita conforme o código abaixo. A função *readQuorum* solicita o valor atual do registrador de todos os membros da visão atual e retorna o valor com o maior *timestamp*, dentre os valores recebidos da maioria dos servidores. A função *writeQuorum* por sua vez faz com que todos os membros da visão

atual escrevam no registrador o valor em *writeMsg*, e retorna com sucesso quando recebe a confirmação de uma maioria. Ambas essas funções tratam o caso em que a visão atual do sistema esteja desatualizada e a fase em execução precisa ser reiniciada.

```
1  // Write v to the system's register.
2  func (cl *Client) Write(v interface{}) error {
3    readValue, err := cl.readQuorum()
4    if err != nil {
5      return err
6    }
7    writeMsg := RegisterMsg{}
8    writeMsg.Value = v
9    writeMsg.Timestamp = readValue.Timestamp + 1
10   writeMsg.ViewRef = cl.ViewRef()
11   err = cl.writeQuorum(writeMsg)
12   if err != nil {
13     return err
14   }
15   return nil
16 }
17 // Read the register value.
18 func (cl *Client) Read() (interface{}, error) {
19   readMsg, err := cl.readQuorum()
20   if err != nil {
21     // Special case:  diffResultsErr
22     if err == diffResultsErr {
23       log.Println("Found divergence: Going to write-back phase of read protocol")
24       return cl.read2ndPhase(readMsg)
25     } else {
26       return nil, err
27     }
28   }
29   return readMsg.Value, nil
30 }
31 func (cl *Client) read2ndPhase(readMsg RegisterMsg) (interface{}, error) {
32   err := cl.writeQuorum(readMsg)
33   if err != nil {
34     return nil, err
35   }
36   return readMsg.Value, nil
37 }
```

4.4. Implementação do Protocolo de Reconfiguração

O protocolo de reconfiguração, ilustrado na Figura 3, pode ser encontrado em [Alchieri et al. 2014]. De forma resumida, os pedidos de *join* e *leave* são armazenados na variável *recv* do tipo *map* (um *mutex* controla acessos concorrentes). A reconfiguração só começa quando um *timeout* ocorre para a visão atual (este parâmetro pode ser configurado no sistema implementado). Neste caso, uma nova sequência de visões é gerada e instalada no sistema.

O gerador de sequências de visões (Seção 4.5) recebe como parâmetro a visão associada e a sequência inicial (que normalmente é a visão atual com as atualizações em *recv*) a ser proposta. Mais de uma sequência pode ser gerada em uma única execução do gerador e são enviadas novamente para o código de reconfiguração por um canal do tipo:

```
1 type newViewSeq struct {
2   ViewSeq     []view.View // sequência de visões gerada
3   AssociatedView view.View // visão associada
4 }
```

Assim que uma sequência é recebida, os servidores instalam a visão mais atual contida na mesma através das seguintes mensagens:

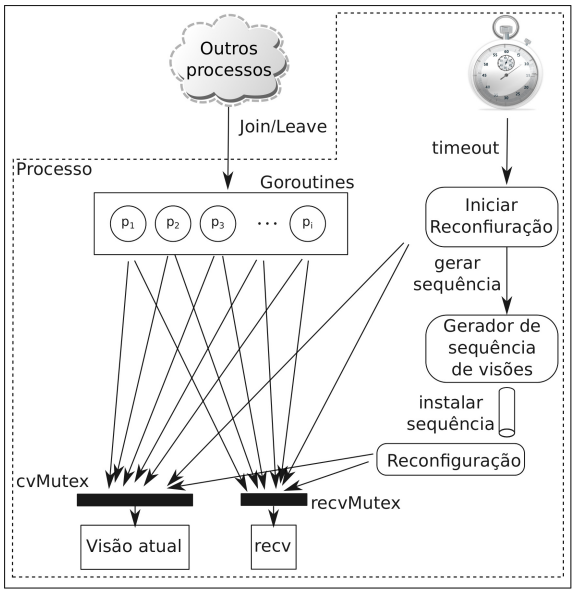


Figura 3. Reconfiguração do sistema

INSTALL-SEQ: É enviada por um servidor que quer instalar uma sequência de visões seq, inicializando este procedimento. As visões em seq são inicializadas uma a uma até que a visão mais atual seja instalada no sistema.

STATE-UPDATE: Os servidores atualizam o estado (valor do registrador e *timestamp*), além do conjunto *recv*, de uma visão instalada através destas mensagens.

VIEW-INSTALLED: É enviada por um servidor para informar que instalou uma visão, liberando os processos que desejam sair do sistema.

Um aspecto interessante do FREESTORE é a separação entre o registrador R/W e a reconfiguração do sistema (Figura 4). Apesar destes protocolos agirem de forma independente, dois *mutexes* (no registrador e na visão atual) devem controlar a execução concorrente de ambos.

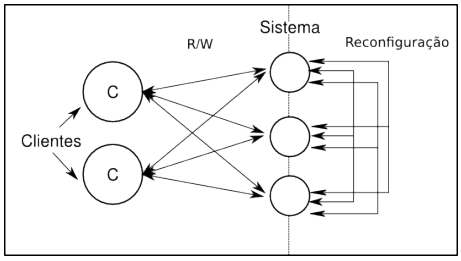


Figura 4. Protocolo de reconfiguração é separado do protocolo de R/W.

4.5. Implementação dos Geradores de Sequência de Visões

Um gerador de sequência de visões captura os requisitos de sincronia necessários entre os servidores para o protocolo de reconfiguração. Esse trabalho implementa dois geradores de sequência de visões. Um faz uso do protocolo de consenso e assim garante que todos os servidores recebam a mesma sequência de visões de seus geradores, enquanto o outro garante a propriedade de que toda sequência gerada contém ou está contida em

uma sequência gerada anteriormente, o que é suficiente para garantir as propriedades de segurança e vivacidade do FREESTORE.

4.5.1. Gerador de Sequência de Visões com Consenso

Esse gerador utiliza um algoritmo de consenso para alcançar acordo sobre a sequência gerada. A solução do consenso implementada foi a do **Paxos** [Lamport 2001]. A implementação considera todos os servidores da visão atual do sistema como *acceptors* e *learners* e funciona da seguinte forma: quando o protocolo de reconfiguração requisita uma sequência, um líder é selecionado para propor uma sequência de visões seq, a qual contém apenas uma visão cujo os membros são definidos pela visão atual mais os pedidos de *join* e *leave* que o líder recebeu desde a última reconfiguração; o protocolo termina com todos decidindo por seq.

4.5.2. Gerador de Sequência de Visões sem Consenso

Este gerador não utiliza consenso, podendo gerar sequências diferentes em diferentes servidores. Através das propriedades de intersecção dos quóruns, uma sequência gerada sempre vai estar contida em outra sequência gerada posteriormente, garantindo as propriedades do sistema. A Figura 5 ilustra o processamento deste gerador. Cada servidor que requisita uma sequência de visões, cria uma instância do gerador que possui sua própria *goroutine*. As mensagens desse gerador são enviadas para o canal da visão associada, onde a *goroutine* correspondente realiza o devido processamento. Para gerar uma sequência de visões, as seguintes mensagens são utilizadas:

SEQ-VIEW: Usada para enviar uma proposta de sequência. Ao receber uma proposta diferente da atual, um servidor atualiza sua proposta computando os novos pedidos de atualização (*updates*) e envia novamente esta mensagem. Ao receber um quórum destas mensagens, um servidor converge para uma sequência.

SEQ-CONV: Um servidor envia esta mensagem para informar que convergiu para uma sequência. Ao receber um quórum destas mensagens, um servidor gera uma sequência para ser instalada no sistema, retornando a mesma para o algoritmo de reconfiguração.

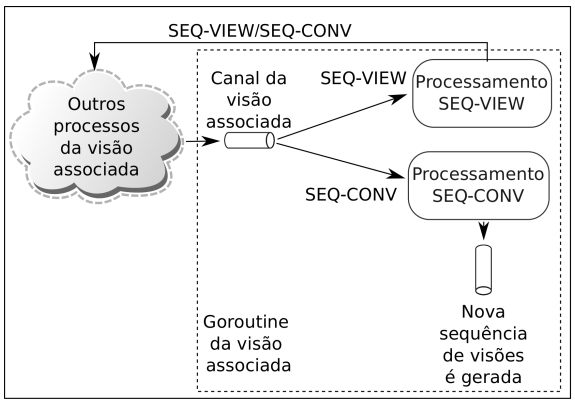


Figura 5. Gerador de sequências de visões sem consenso.

4.6. Inicialização de Servidores e Clientes

Um servidor do sistema, quando inicializado, realiza as seguintes principais preparações:

- **Inicializa *listener*:** Variável que contém a conexão do servidor com a rede.
- **Inicializa *register*:** Variável que representa o registrador, sendo inicializada para um valor inicial nulo com *timestamp* 0. Um *mutex* bloqueia o acesso ao registrador, apenas sendo desbloqueado quando o servidor instala uma visão a que pertence, i.e., habilita a execução de operações de *R/W*.
- **Inicializa *currentView*:** Variável que contém a atual visão instalada no sistema, sendo inicializada da seguinte forma:
  - **Servidor pertence à visão inicial do sistema:** O servidor simplesmente instala a visão inicial do sistema.
  - **Servidor não pertence à visão inicial do sistema:** O servidor requisita a visão atual do sistema a outro servidor do sistema. O endereço desse servidor é passado na sua inicialização.
- **Outros:** Os módulos responsáveis pela escuta de novas conexões são inicializados; as estruturas principais são inicializadas (ex.: canais e *maps*); as *goroutines* referentes ao protocolo de reconfiguração (ex.: *goroutine* que mantém o timer que dispara uma nova reconfiguração) que sempre estão ativas são inicializadas.

Por outro lado, quando um cliente é inicializado, o mesmo apenas realiza a inicialização da *currentView* requisitando a um servidor do sistema a visão atual. Após isso, tal servidor está apto a realizar operações de *R/W*.

5. Experimentos

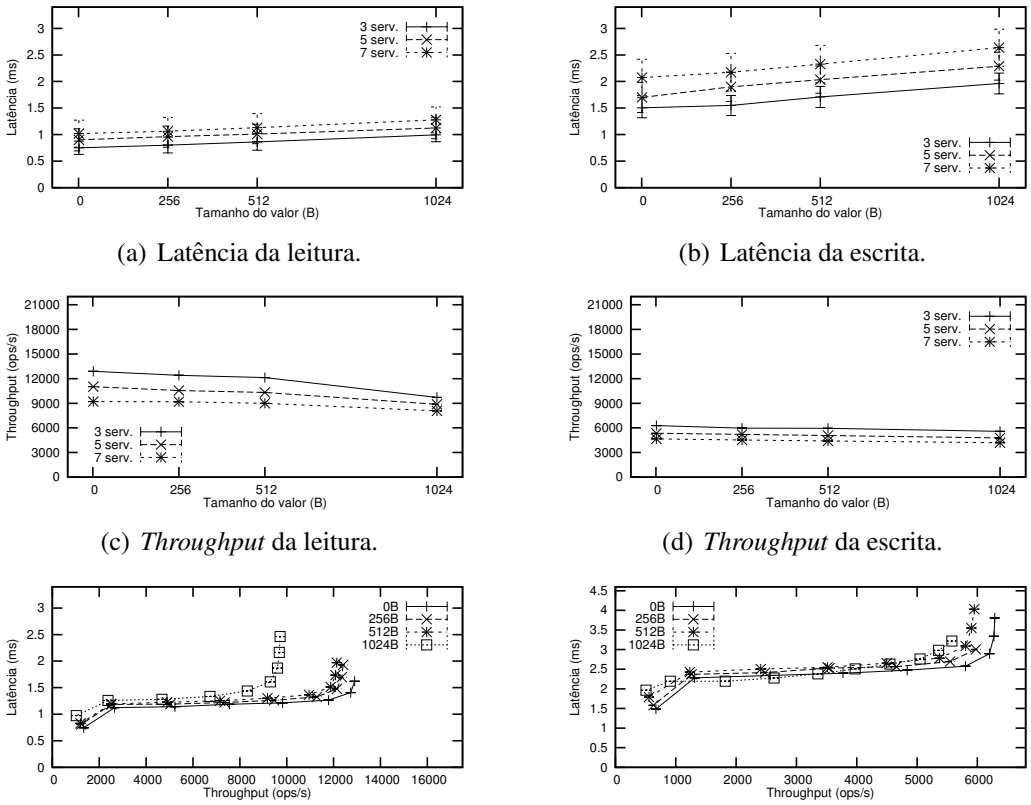
Visando analisar o desempenho da implementação desenvolvida, alguns experimentos foram realizados no Emulab [White et al. 2002], em um ambiente consistindo por até 20 máquinas *pc3000* (3.0 GHz 64-*bit* Pentium Xeon com 2GB de RAM e interface de rede gigabit) conectadas a um *switch* de 100Mb. Cada servidor executou em uma máquina separada, enquanto que até 26 clientes foram distribuídos uniformemente nas máquinas restantes (2 por máquina). O ambiente de *software* utilizado foi o sistema operacional Fedora 15 64-*bit* com *kernel* 2.6.20 e compilador *Go* 1.2.

Os experimentos realizados foram: (1) alguns *micro-benchmarks* para avaliar a latência e o *throughput*; (2) uma comparação entre o modelo estático e o modelo dinâmico; e (3) um experimento que mostra o comportamento do sistema na presença de falhas e reconfigurações. Tanto a latência quanto o *throughput* foram medidos nos clientes, sendo que o *throughput* foi calculado somando-se quantas operações o conjunto de clientes conseguiu executar.

***Micro-benchmarks*.** Começamos pela análise de uma série de *micro-benchmarks* comumente utilizados para avaliar estes sistemas, os quais consistem na leitura e escrita de valores com diferentes tamanhos: 0, 256, 512 e 1024 *bytes*.

As figuras 6(a) e 6(b) mostram a latência da leitura e escrita, respectivamente, para o sistema configurado com 3,5,e 7 servidores (i.e., tolerando 1,2 ou 3 falhas), enquanto que as figuras 6(c) e 6(d) apresentam os valores de *throughput* para as mesmas configurações. Já as figuras 6(e) e 6(f) apresentam a relação entre o *throughput* e a latência de leitura e escrita, respectivamente, para o sistema configurado com 3 servidores.

Neste primeiro conjunto de experimentos, podemos perceber que tanto o aumento no número de servidores quanto no tamanho do valor a ser lido/escrito faz com que o desempenho do sistema diminua. Este comportamento era esperado e deve-se ao fato de que com mais servidores um maior número de mensagens são enviadas e esperadas para atingir um quórum, enquanto que o aumento no tamanho do valor faz com que estas mensagens também tenham um tamanho maior. Finalmente, podemos perceber que os custos de uma leitura são praticamente a metade dos custos de uma escrita. Isso deve-se ao protocolo de leitura e escrita empregado [Attiya et al. 1995], que requer dois passos de comunicação para leituras e quatro passos para escritas, considerando execuções sem concorrência entre leituras e escritas.



(a) Latência da leitura.

(b) Latência da escrita.

(c) *Throughput* da leitura.

(d) *Throughput* da escrita.

(e) Leitura ($n = 3$ servidores).

(f) Escrita ($n = 3$ servidores).

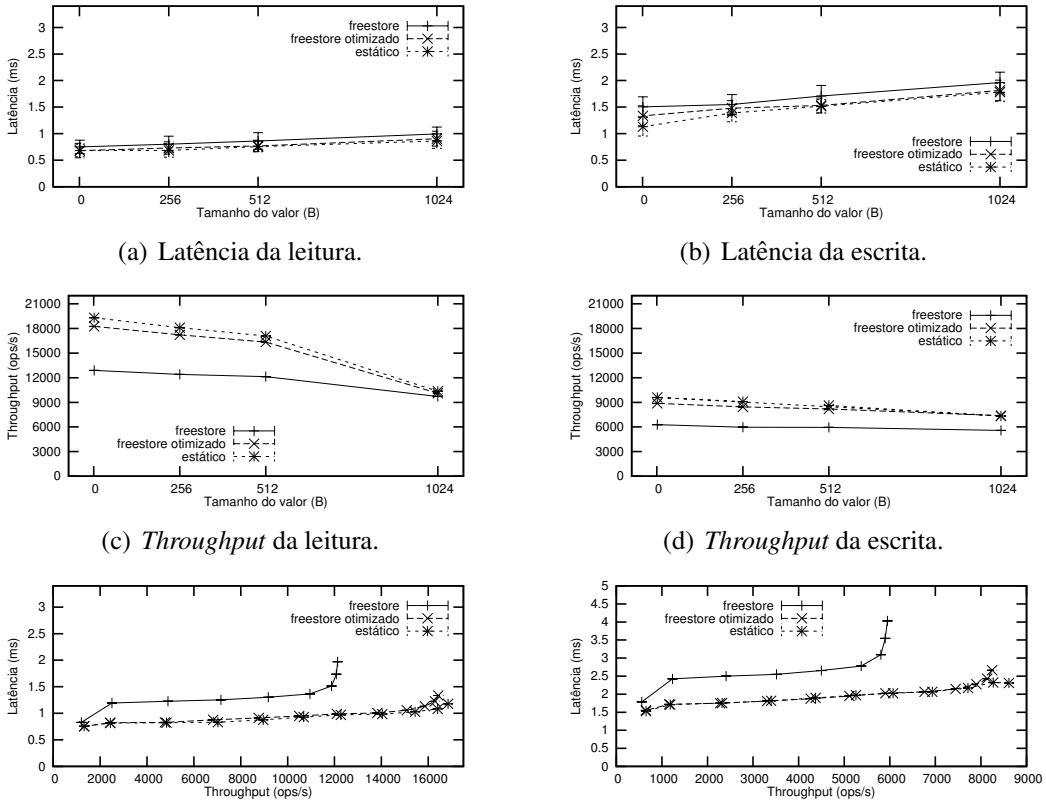
Figura 6. Desempenho do sistema.

Dinâmico vs. Estático. Este segundo experimento visa comparar o desempenho do sistema implementado com a sua versão para ambientes estáticos (onde o conjunto de servidores é fixo). Basicamente, os custos adicionais na versão dinâmica são o envio da visão atual conhecida pelo cliente em cada mensagem e a verificação se a mesma é a visão mais atualizada do servidor. Reportamos também os resultados para uma implementação otimizada do FREESTORE, onde apenas um *hash* criptográfico (SHA-1) das visões é anexado nas mensagens, diminuindo o tamanho das mesmas.

As figuras 7(a) e 7(b) apresentam os valores para a latência de leitura e escrita, respectivamente, para o sistema configurado com 3 servidores, enquanto que as figuras 7(c)

e 7(d) apresentam os valores de *throughput*. Já as figuras 7(e) e 7(f) apresentam a relação entre o *throughput* e a latência de leitura e escrita, respectivamente, considerando um valor de 512 *bytes*.

Nestes experimentos podemos perceber que a versão otimizada do FREESTORE apresenta um desempenho praticamente igual a sua versão estática. Conforme já comentado, a modularidade do FREESTORE faz com que os protocolos de reconfiguração praticamente não interfiram nos protocolos de leitura e escrita, quando executados em períodos onde não estão ocorrendo reconfigurações. Pelos mesmos motivos anteriormente descritos, as escritas apresentaram praticamente a metade do desempenho das leituras.



(a) Latência da leitura.

(b) Latência da escrita.

(c) *Throughput* da leitura.

(d) *Throughput* da escrita.

(e) Leitura (valor de 512 *bytes*).

(f) Escrita (valor de 512 *bytes*).

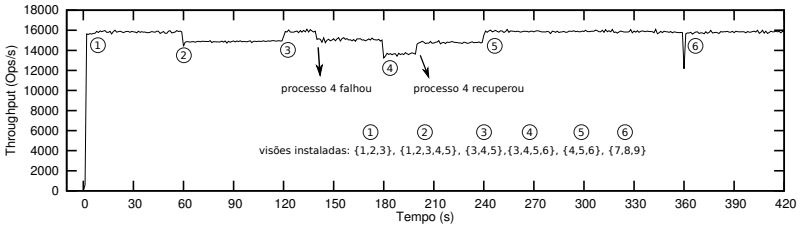
Figura 7. Dinâmico vs. Estático ($n = 3$ servidores).

Reconfigurações e Falhas. Este último experimento visa analisar o comportamento do sistema em execuções com reconfigurações e falhas de processos. Para isso, o sistema foi inicialmente configurado com 3 servidores (visão inicial = $\{1,2,3\}$) e os clientes executaram operações de leitura de um valor com tamanho de 512 *bytes*. O período entre reconfigurações foi configurado em 60 segundos, i.e., a cada 60 segundos uma reconfiguração é executada.

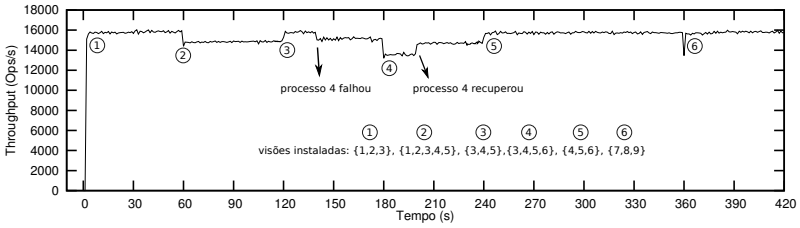
As figuras 8(a) e 8(b) mostram o *throughput* apresentado pelo sistema durante a sua execução utilizando ou não um protocolo de consenso para reconfigurações, respectivamente. Os eventos que ocorreram durante a execução foram os seguintes:

- 0s - início do experimento.

- 30s - o servidor 4 envia um pedido de *join* e aguarda sua entrada no sistema.
- 40s - o servidor 5 envia um pedido de *join* e aguarda sua entrada no sistema.
- 60s - uma reconfiguração adiciona os servidores 4 e 5 no sistema, instalando a visão {1,2,3,4,5}.
- 80s - o servidor 1 envia um pedido de *leave* e aguarda sua saída do sistema.
- 100s - o servidor 2 envia um pedido de *leave* e aguarda sua saída do sistema.
- 120s - uma reconfiguração remove os servidores 1 e 2 do sistema, instalando a visão {3,4,5}.
- 140s - o servidor 4 sofre um *crash* (falha).
- 160s - o servidor 6 envia um pedido de *join* e aguarda sua entrada no sistema.
- 180s - uma reconfiguração adiciona o servidor 6 no sistema, instalando a visão {3,4,5,6}.
- 200s - o servidor 4 se recupera e volta ao sistema.
- 220s - o servidor 3 envia um pedido de *leave* e aguarda sua saída do sistema.
- 240s - uma reconfiguração remove o servidor 3 do sistema, instalando a visão {4,5,6}.
- 300s - o sistema tenta executar uma reconfiguração, mas como não havia nenhum pedido, nada é processado.
- 320s - os servidores 4,5 e 6 enviam um pedido de *leave* e aguardam pelas suas saídas do sistema.
- 340s - os servidores 7,8 e 9 enviam um pedido de *join* e aguardam pelas suas entradas no sistema.
- 360s - uma reconfiguração removes os servidores 4,5 e 6 e adiciona os servidores 7,8 e 9 no sistema, instalando a visão {7,8,9}.
- 420s - fim do experimento.



(a) Com consenso.



(b) Sem consenso.

Figura 8. Reconfigurações e falhas.

Neste experimento podemos perceber que o desempenho do sistema diminui quando mais servidores estiverem presentes no mesmo. Além disso, devido às características de desacoplamento e modularidade do FREESTORE, o desempenho é praticamente o mesmo para ambas as abordagens de reconfiguração (com ou sem consenso).

De fato, o tempo médio de uma reconfiguração foi de 19ms na abordagem sem consenso (como os pedidos de reconfiguração foram recebidos por todos os processos antes do início da reconfiguração, todos tinham a mesma proposta e o protocolo do gerador converge com apenas um passo – *SEQ-VIEW*, além disso este tempo também engloba os passos do algoritmo de reconfiguração), bloqueando as operações de R/W por apenas 4ms. Para a solução com consenso, o tempo médio de uma reconfiguração foi de 40ms (tempo para executar o protocolo de consenso e os passos de algoritmo de reconfiguração), bloqueando as operações de R/W por 4ms. Note que o tempo em que as operações de R/W ficaram bloqueadas foi o mesmo em ambas as abordagens, pois este bloqueio somente ocorre depois de uma sequência ser gerada (i.e., após a execução do gerador).

6. Conclusões

Este artigo discutiu uma implementação para os protocolos do FREESTORE e através de um conjunto de experimentos possibilitou uma maior compreensão sobre o comportamento destes protocolos quando inseridos nos mais variados cenários de execução, possibilitando também uma maior entendimento sobre o processo de reconfiguração de memória compartilhada. Como trabalhos futuros, pretendemos implementar os protocolos do DynaStore [Aguilera et al. 2011] e comparar o desempenho destes sistemas.

Referências

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

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

Alchieri, E., Bessani, A., Greve, F., and Fraga, J. (2014). Reconfiguração modular de sistemas de quóruns. In *Anais do 32º Simpósio Brasileiro de Redes de Computadores*.

Attiya, H., Bar-Noy, A., and Dolev, D. (1995). Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142.

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

Dwork, C., Lynch, N. A., and Stockmeyer, L. (1988). Consensus in the presence of partial synchrony. *Journal of ACM*, 35(2):288–322.

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

Gifford, D. (1979). Weighted voting for replicated data. In *7th ACM Symposium on Operating Systems Principles*.

Gilbert, S., Lynch, N., and Shvartsman, A. (2010). Rambo: A robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing*, 23(4).

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

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

Lamport, L. (2001). Paxos Made Simple. *ACM SIGACT News*, 32(4):18–25.

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

Martin, J.-P. and Alvisi, L. (2004). A Framework for Dynamic Byzantine Storage. In *34th International Conference on Dependable Systems and Networks*.

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 Symposium on Operating Systems Design and Implementations*.

Serviço de Líder em Sistemas Dinâmicos com Memória Compartilhada

Cátia Khouri[1,2], **Fabíola Greve**[2]

[1]Departamento de Ciências Exatas – Univ. Estadual do Sudoeste da Bahia
Vitória da Conquista, BA – Brasil

[2]Departamento de Ciência da Computação – Univ. Federal da Bahia (UFBA)
Salvador, BA – Brasil

catia091@dcc.ufba.br, fabiola@dcc.ufba.br

Abstract. *In spite of the importance of dynamic distributed environments, such as SANs (storage area networks) and multicore architectures, we found very few proposals of models and protocols for implementing eventual leader election. This abstraction is fundamental to the implementation of consistency and fault-tolerance requirements. Most approaches for electing a leader consider static systems, where processes communicate by message passing, satisfying timing constraints. This paper presents a leader election service, of class* Ω*, for a* dynamic *asynchronous system subject to crash failures, where processes communicate through* atomic shared registers *following a memory access pattern, free of temporal requirements.*

Resumo. *Apesar da importância que ambientes dinâmicos como as SANs (redes de área de armazenamento) e as arquiteturas multi-núcleo ocupam no cenário atual dos sistemas distribuídos, poucas são as propostas de modelos e protocolos para a implementação de eleição de líder após um tempo nesses contextos. Essa abstração é fundamental para a implementação dos requisitos de consistência e tolerância a falhas desses sistemas. Entretanto, a maioria das abordagens de eleição de líder considera sistemas estáticos, onde os processos se comunicam por troca de mensagens, satisfazendo requisitos temporais. Este artigo apresenta um serviço de eleição de líder, da classe* Ω*, para um* sistema dinâmico *assíncrono, sujeito a falhas por parada, em que os processos se comunicam através de* registradores atômicos compartilhados *e segundo um padrão de acesso à memória, livre de requisitos temporais.*

1. Introdução

Garantir tolerância a falhas em sistemas distribuídos assíncronos não é uma tarefa trivial. Na realidade, é mesmo impossível distinguir se um processo falhou ou se apenas está muito lento quando não existem limites definidos para a velocidade de passos de processos ou para o tempo de transmissão de mensagens. Para contornar essa dificuldade, uma alternativa é estender o sistema com a utilização de detectores de falhas não confiáveis [Chandra and Toueg 1996]. Estes mecanismos funcionam como oráculos distribuídos que fornecem informações sobre processos falhos no sistema.

Um desses detectores largamente utilizado é o *detector de líder após um tempo* (do inglês: *eventual leader detector*), conhecido como Ômega (ou Ω) [Chandra et al. 1996].

Num sistema distribuído confiável, camadas superiores de serviço tais como consenso, difusão atômica, replicação de máquinas de estado, etc., contam com um serviço de líder após um tempo para prover os processos com uma primitiva que retorna a identidade de um líder, i.e., um processo correto no sistema. Embora possa fornecer informações equivocadas durante um intervalo de tempo finito, após o período de instabilidade, todo processo do sistema recebe a mesma identidade de líder. Diz-se então que todo processo no sistema confia naquele eleito como líder.

Um atributo importante de um detector Ω é o de permitir o projeto de algoritmos *indulgentes* [Guerraoui and Raynal 2003]. Nesse caso, os algoritmos podem conviver pacificamente com os erros do detector, garantindo que suas propriedades de segurança (ou *safety*) não sejam violadas durante o período instável. A terminação (ou *liveness*) de tal algoritmo será então viabilizada pelo detector quando todos os processos confiarem no mesmo líder.

A classe de detectores Ω é a mais fraca a possibilitar a resolução do consenso em sistemas assíncronos sujeitos a falhas [Chandra et al. 1996]. Entretanto, a implementação desse detector em si também não é possível em sistemas puramente assíncronos (pelas razões colocadas acima). É preciso enriquecer tal sistema com suposições a respeito do comportamento dos processos e/ou canais de comunicação. Nesse sentido, vários trabalhos têm sido publicados que propõem propriedades adicionais a sistemas assíncronos e os algoritmos correspondentes que implementam o serviço de líder nessas condições. Contudo, a maioria desses trabalhos é voltada para sistemas em que o conjunto de participantes é estático, conhecido, e a comunicação se dá através de troca de mensagens.

Este artigo trata da implementação de um serviço de líder em um *sistema dinâmico* assíncrono sujeito a falhas por parada (*crash*), em que os processos se comunicam através de uma *memória* constituída de *registradores atômicos compartilhados*. A nossa motivação vem da proliferação de sistemas como redes de dispositivos móveis *ad-hoc*, computação em nuvem, sistemas p2p, redes de área de armazenamento (SAN's), etc. Nesses sistemas, considerados dinâmicos, os nós podem entrar e sair arbitrariamente, de modo que o conjunto de participantes é auto-definido e variável com a execução. Normalmente, no início de uma execução, cada nó conhece apenas sua identidade, e sabe que as identidades dos demais nós são distintas. Esse dinamismo introduz uma nova fonte de imprevisibilidade no sistema e deve ser levado em conta no projeto dos algoritmos.

O modelo de interação por memória compartilhada adequa-se bem a sistemas em que discos conectados em rede (*network attached disks*) aceitam requisições de leitura e escrita propiciando a comunicação entre computadores. Esses discos constituem uma SAN (*storage area network*), que implementa uma abstração de memória compartilhada e tem se tornado cada vez mais frequente como mecanismo de tolerância a falhas. Os trabalhos em [Gafni and Lamport 2003, Guerraoui and Raynal 2007] apresentam algoritmos para tais sistemas baseados no detector de líder Ω. Adicionalmente, o desenvolvimento crescente de modernas arquiteturas multi-núcleo tem impulsionado estudos em sistemas de memória compartilhada assíncronos para os quais Ω tem sido aplicado como gerenciador de contenção, como em [Aguilera et al. 2003, Fernández et al. 2010].

Apesar da importância que tais ambientes ocupam no cenário atual dos sistemas distribuídos, poucas são as propostas de modelos e protocolos para a implementação de

Ω nessas circunstâncias. A maioria das abordagens considera sistemas estáticos de passagem de mensagens que fazem suposições adicionais relativas à sincronia de processos e/ou canais de comunicação. Em geral, essas propostas sugerem que todos ou parte dos canais funcionam "em tempo" (*timely*) a partir de um determinado instante e variam essencialmente no teor de sincronia que exigem para convergir. Outra abordagem, menos frequente, baseia-se num *padrão de mensagens* ao invés de assumir limites para processos ou transferência de mensagens. Ao contrário da primeira, protocolos baseados nesta última abordagem não utilizam temporizadores e são por isso chamados livres de tempo (ou *time-free*). Poucos são os protocolos para sistemas dinâmicos de memória compartilhada e, até onde sabemos, nenhum baseia-se em alguma abordagem livre de tempo.

Nesse artigo apresentamos um protocolo de detecção de líder após um tempo, da classe Ω, para um sistema dinâmico assíncrono, onde processos se comunicam através de memória compartilhada, seguindo um padrão de acesso à memória e portanto uma abordagem livre de tempo. Nesse sentido, nosso trabalho é pioneiro.

2. Modelo de Sistema para Eleição de Líder

O sistema dinâmico é composto por um conjunto infinito Π de processos que se comunicam através de uma memória compartilhada e que podem falhar (*crashing*), parando prematura ou deliberadamente (por exemplo, saindo arbitrariamente do sistema). Não fazemos suposições sobre a velocidade relativa dos processos, para dar passos ou para realizar operações de acesso à memória compartilhada, isto é, o sistema é assíncrono. Para facilitar a apresentação, assumimos a existência de um relógio global que não é acessível pelos processos. A faixa de ticks do relógio, $\mathcal{T}$, é o conjunto dos números naturais.

2.1. Processos

Consideramos o *modelo de chegada finita* [Aguilera 2004], isto é, o sistema tem um número infinito de processos que podem entrar e sair em instantes arbitrários, mas em cada execução r o conjunto de participantes $\Pi = \{p_1, ..., p_n\}$ é limitado. Π (inclusive n) pode variar de execução a execução, mas existe um instante a partir do qual nenhum novo processo entra (por isso o nome "*chegada finita*"). Cada processo localiza-se em um nó distinto e cada um deles possui uma identidade distinta tal que o conjunto dessas identidades é totalmente ordenado (denotamos i a identidade do processo p_i). Um processo p_i que deseja entrar no sistema não têm conhecimento sobre o conjunto de participantes e o estado dos registradores compartilhados e nem mesmo possui uma identidade.

Em um sistema de passagem de mensagens em que o conjunto de participantes é desconhecido, os processos podem, por exemplo, começar a executar o algoritmo de Ω e à medida que o algoritmo progride, os nós aprendem dinamicamente sobre a existência dos outros participantes, através de trocas de mensagens, construindo assim o *membership*. No nosso modelo de memória compartilhada, abstraímos essa construção do conjunto de participantes delegando essa tarefa à implementação da memória compartilhada subjacente. Assim, para entrar no sistema, p_i invoca uma operação *join()* a fim de inteirar-se do estado do sistema, criar seus registradores e obter uma identidade (distinta da dos demais participantes). Além disso, como será visto mais adiante, a operação $join()$ se encarrega de garantir que os processos que já se encontram no sistema criem registradores que armazenarão informações sobre p_i com um estado inicial que favoreça a manutenção de um

líder correto (estável) eleito. Um processo então só pode começar a participar ativamente da computação e concorrer à posição de líder após o término da operação $join()$.

Um processo que segue a especificação do algoritmo e não falha em uma execução é dito *correto*. O padrão de falhas do sistema, $F(t)$, é o conjunto de processos que entraram no sistema, mas falharam até o instante t, e um processo pertencente a $F(t)$, para algum t, é dito *faltoso*. Considerando a natureza dinâmica do sistema, denotamos $ATIVO(t)$ o conjunto de processos que entraram no sistema e não saíram (nem falharam) até o instante t. Se um processo sai (ou falha) e volta para a computação, ele é considerado um novo processo e portanto terá uma nova identidade.

A despeito da movimentação de entrada e saída de processos, para que se possa realizar alguma computação útil, é preciso que exista um período de estabilização do sistema bem como um conjunto de processos que permaneça ativo neste período. Definimos então o conjunto de processos *estáveis* a partir da definição do conjunto de processos ativos. Informalmente, um processo correto que nunca deixa a computação, ou seja, um processo que se ligou ao sistema num instante t e permanece ativo $\forall t' \geq t$ é dito *estável*.

Definição 1 (STABLE) *STABLE* = $\{p_e : \exists t \in \mathcal{T}, \forall t' \geq t : p_e \in ATIVO(t')\}$

Para garantir progresso, o tamanho do conjunto de processos estáveis deve ser limitado inferiormente. No contexto deste trabalho, uma vez que não se conhece *a priori* nem n (cardinalidade do sistema) nem f (máximo número de processos falhos), utilizaremos o parâmetro α para denotar o número mínimo de processos que devem estar vivos no período estável. Assim, uma condição necessária para a terminação (*liveness*) neste modelo é: $|STABLE| \geq \alpha$, onde α abstrai qualquer par (n, f). No modelo clássico de sistemas distribuídos, α corresponde ao valor $(n - f)$.

2.2. Memória Compartilhada

Um sistema de memória compartilhada distribuída é uma abstração construída no topo de um sistema de passagem de mensagens que permite aos processos comunicarem-se invocando operações sobre objetos compartilhados. Um objeto compartilhado possui um *tipo*, que define o domínio de valores que podem ser armazenados; e um conjunto de operações que são a única forma de acessar o objeto. No modelo em questão esses objetos são *registradores atômicos* multivalorados do tipo 1-*escritor/n-leitores* (1W/nR), que se comportam corretamente. Um registrador multivalorado armazena um valor inteiro e provê duas operações básicas: $read$, que retorna o valor armazenado; e $write$, que escreve (armazena) um valor no registrador.

Um registrador compartilhado modela uma forma de comunicação persistente onde o emissor é o escritor, o receptor é o leitor, e o estado do meio de comunicação é o valor do registrador. Comportamento *correto* significa que o registrador sempre pode executar uma leitura ou escrita e nunca corrompe seu valor. 1W/nR significa que o registrador pode ser escrito por um processo (seu proprietário) e lido por múltiplos processos.

Em um sistema com registradores atômicos, embora as operações sobre um mesmo registrador possam se sobrepor em um intervalo de tempo, elas parecem ser instantâneas, de modo que para qualquer execução existe alguma forma de ordenar totalmente leituras e escritas como se elas ocorressem instantaneamente em algum ponto entre sua invocação e resposta. Assim, o valor retornado por uma leitura que sobrepõe uma

ou mais escritas é o mesmo que seria retornado se não houvesse sobreposição. Dizemos então que um registrador atômico é *linearizável* [Herlihy and Wing 1990].

3. Especificação de Ω

Já é bem estabelecido que não existe solução determinística para alguns problemas fundamentais em sistemas distribuídos assíncronos sujeitos a falhas [Fischer et al. 1985]. Informalmente, essa impossibilidade é explicada pela dificuldade de se distinguir, em determinado instante, se um processo parou ou está excessivamente lento. Para contornar essa dificuldade, uma alternativa é estender o sistema com mecanismos detectores (de falhas ou de líder) tais como os propostos por [Chandra and Toueg 1996]. Tais dispositivos são constituídos de módulos distribuídos que têm o objetivo de prover o sistema com "dicas" sobre falhas de processos (ou líderes).

Um serviço *detector de líder após um tempo* da classe Ω provê os processos com uma função *leader()* que, quando invocada por um processo p_i, fornece a identidade de um processo p_j que o detector considera correto naquele instante. Após um tempo, essa identidade é única para todos os processos que invocaram a função. Num contexto de sistema dinâmico, um detector Ω satisfaz assim à seguinte propriedade:

Liderança após um tempo (*eventual leadership*): existe um instante após o qual qualquer invocação de $leader()$ por qualquer processo $p_i \in STABLE$ retorna o mesmo processo $p_l \in STABLE$.

4. Eleição de Líder em um Sistema Dinâmico de Memória Compartilhada

Um sistema dinâmico, no qual processos podem entrar e sair arbitrariamente, apresenta uma dificuldade adicional no projeto de algoritmos tolerantes a falhas. Por exemplo, numa implementação de Ω pode acontecer de nenhum processo permanecer por tempo suficiente para ser eleito. Desse modo, é preciso que o sistema satisfaça alguma propriedade de estabilidade para que a computação progrida e então termine.

A abordagem utilizada neste trabalho é baseada num *padrão de acesso à memória* compartilhada, de certo modo semelhante à introduzida em [Mostefaoui et al. 2003] para sistemas de passagem de mensagem. Informalmente, assumimos que existe um conjunto de α processos que permanecem vivos por um período de tempo suficiente para eleger o líder. Esse valor de α corresponde ao valor $(n - f)$, em sistemas estáticos, que representa um limite inferior para o número de processos que precisam interagir para garantir progresso e terminação a certos protocolos.

4.1. Propriedade do Sistema

No algoritmo a ser apresentado, com o intuito de informar aos demais processos do sistema que está ativo, o processo p_i se mantém incrementando um contador num registrador compartilhado, de nome $Alive[i]$. Por outro lado, periodicamente, os demais processos p_j consultam o valor em $Alive[i]$ para saber se p_i progrediu. Assim, $\forall p_j, \forall p_i$: p_j lê $Alive[i]$ e, caso o valor lido seja igual àquele lido anteriormente, então p_j pune p_i, suspeitando que ele não esteja mais ativo no sistema. Chamaremos $vencedores_j(t)$ ao conjunto dos primeiros α processos cujos registradores p_j percebeu estarem atualizados com relação à sua última leitura. Formalizamos então a propriedade que habilita o progresso e término do algoritmo de líder a ser apresentado.

Propriedade 1 (Padrão de Acesso à Memória – PAM) *Existe um instante t e um processo $p_l \in STABLE$ tais que, $\forall t' \geq t$, $\forall p_j \in STABLE : p_l \in vencedores_j(t')$.*

Algorithm 1 EVENTUAL LEADER ELECTION (*code for p_i*)

INITIALIZATION
(1) $join()$;
(2) start TASK 0, TASK 1, TASK 2

TASK 0: INIT-REGS
UPON REQUEST $join()$ FROM p_j
(3) **write**$(Punishments[i][j], Punishments[i][leader()] + 1)$;

TASK 1: ALIVE
(4) **repeat forever**
(5) $alive_i \leftarrow alive_i + 1$;
(6) **write** $(Alive[i], alive_i)$;

TASK 2: PUNISHMENT
(7) **repeat forever**
(8) $updated_i \leftarrow \{i\}$;
(9) **while** $(|updated_i| < \alpha)$ **do**
(10) **for each** (j) **do**:
(11) **if** $(j \notin updated_i)$ **then** {
(12) **read** $(Alive[j], alive_i^j)$;
(13) **if** $(alive_i^j \neq last_alive_i[j])$ **then** {
(14) $last_alive_i[j] \leftarrow alive_i^j$;
(15) $updated_i \leftarrow updated_i \cup j$; } }
(16) **for each** $(j \notin updated_i)$ **do**:
(17) **read** $(Punishments[i][j], pun2_i)$;
(18) **write**$(Punishments[i][j], pun2_i + 1))$;

$leader()$
(19) **for each** j **do**:
(20) $soma_i[j] \leftarrow 0$;
(21) **for each** k **do**:
(22) **read**$(Punishments[k][j], pun1_i)$;
(23) $soma_i[j] \leftarrow soma_i[j] + pun1_i$;
(24) $leader_i \leftarrow Min(soma_i[j], j)$;
(25) return $leader_i$;

4.2. Visão Geral do Algoritmo de Eleição de Líder

O Algoritmo 1 é executado em paralelo por todos os processos. Um processo p_i define como seu líder o processo que ele considera "menos suspeito". Essa ideia de suspeição é expressa através de contadores indexados. Sempre que p_i suspeita de p_j, incrementa o contador correspondente. Assim, define-se um arranjo de registradores tal que $Punishments[i][j]$ representa o número de vezes que p_i suspeitou (e puniu) p_j.

O nível de suspeição de um processo p_i com relação a um processo p_j será definido pela soma dos contadores: $soma_i[j] = \sum_{\forall k} Punishments[k][j]$. O processo escolhido como líder será o p_j para o qual $soma_i[j]$ seja mínima. Se houver mais de uma soma com valor igual ao mínimo, o desempate será definido pela menor identidade de processo, isto é, será considerada a ordem lexicográfica. Dados os pares $(soma_i[x], p_x)$ e $(soma_i[y], p_y)$ com $x < y$, $Min((soma_i[x], p_x), (soma_i[y], p_y)) = p_x$, se $soma_i[x] \leq soma_i[y]$; e p_y , se $soma_i[y] < soma_i[x]$.

Quando um processo p_r quer entrar no sistema, invoca a operação $join()$ que provê um estado consistente dos registradores para p_r; inicializa os registradores $Alive[r]$ e $Punishments[r][i]$, $\forall i$; e aloca espaço para seus arranjos dinâmicos. Os demais processos p_i, ao receberem um solicitação de $join()$ da parte de p_r, criam seus registradores $Punishments[i][r]$ devidamente inicializados; e alocam espaço em seus arranjos dinâmicos locais para acomodar p_r. Apenas após o retorno de $join()$ um processo p_r estará apto a participar da computação e será considerado pelos demais processos no sistema como um candidato a líder (incluindo os registradores correspondentes na escolha).

Para impedir que um líder estável seja demovido por um recém-chegado, cada registrador $Punishments[i][r]$ é inicializado com o valor do contador de punições de p_i atribuído a p_l acrescido de 1: $Punishments[i][r] \leftarrow Punishments[i][l] + 1$. Dessa forma, o nível de suspeição inicial do recém-chegado ($\sum_{\forall i} Punishments[i][r]$) será maior que o do líder corrente.

No algoritmo são utilizados os seguintes registradores compartilhados:

- $Alive[i]$ – contador responsável por indicar o progresso de p_i, inicializado com 0.
- $Punishments[i][j]$ – contador pertencente a p_i que registra o número de punições aplicadas por p_i a p_j. Inicializado com $Punishments[i][l]+1$ (o contador referente ao líder corrente acrescido de 1).

Um processo p_i também utiliza as seguintes variáveis locais:

- $alive_i$ – contador que controla o progresso de p_i cujo valor é escrito em $Alive[i]$. Inicializado com 0.
- $alive_i^j$ – variável auxiliar usada para leitura de $Alive[j]$.
- $last_alive_i[1..]$ – vetor dinâmico que armazena o último valor lido em $Alive[1..]$. Cada entrada é inicializada com 0.
- $soma_i[1..]$ – vetor dinâmico que armazena a soma dos contadores de punições de cada p_j.
- $updated_i$ – conjunto utilizado para controlar quais processos tiveram seu progresso verificado.
- $pun1_i$, $pun2_i$ – variáveis auxiliares usadas para leitura de $Punishments[i][j]$.

4.3. Descrição do Algoritmo

Para ligar-se ao sistema, p_i invoca a operação $join()$ (linha 1), e ao retornar inicia três tarefas em paralelo (linha 2).

TASK 0: INIT-REGS – Ao perceber uma requisição de *join()* de algum p_j, p_i inicializa seu registrador $Punishments[i][j]$ com $Punishments[i][l] + 1$ (linha 3).

TASK 1: ALIVE – Ao executar essa tarefa, p_i informa intermitentemente aos demais participantes que ele está vivo com o fim de evitar falsas suspeitas a seu respeito. Ele faz isso incrementando seu registrador compartilhado $Alive[i]$ (linhas 4-6).

TASK 2: PUNISHMENT – Enquanto executa esta tarefa, p_i se mantém verificando se os registradores $Alive[j]$ foram atualizados desde sua última leitura (linhas 7-15). Para fazer isso, a cada iteração do laço mais externo (linha 7), p_i guarda o valor lido em cada $Alive[j]$ no vetor local $last_alive_i[j]$, a fim de poder compará-lo com a leitura na próxima iteração. Como p_i nunca pune a si mesmo, inicialmente ele atribui sua própria identidade à variável conjunto local $updated_i$ que ele utiliza para controlar quais registradores são achados atualizados (linha 8). Então ele procura por pelo menos α processos p_j que tenham atualizado seus registradores (linhas 9-10). Para cada um desses (linhas 10-13), p_i guarda o valor atualizado de $Alive[j]$ e inclui p_j em $updated_i$ (linhas 14-15).

A partir daí, p_i pune cada processo p_j que não foi encontrado "atualizado". Para isso ele adiciona 1 ao valor corrente no registrador $Punishments[i][j]$ (linhas 16-18). Assim, os primeiros α processos que atualizarem seus registradores $Alive[j]$ não são punidos.

leader() – Esta função retorna uma identidade de processo que, do ponto de vista de p_i é correto e considerado líder por ele. Para a escolha do líder, a soma das punições atribuídas a cada p_j é acumulada em $soma_i[j]$ (linhas 19-23). É escolhido como líder o processo com menor soma, utilizando a identidade como desempate (linhas 24-25).

4.4. Prova de Correção

Lema 1 *Existem um instante t e um processo $p_l \in STABLE$ tais que após t, p_l não será jamais punido por qualquer processo p_j.*

Prova. As punições ocorrem em TASK 2: PUNISHMENT. O comportamento de p_j vai depender do fato dele pertencer ou não ao conjunto $STABLE$. (1) Se $p_j \in STABLE$, de acordo com a Propriedade 1, existem um processo p_l e um instante t', tais que $\forall t'' \geq t'$, p_j encontrará $Alive[l]$ atualizado com relação à sua última leitura e portanto avaliará o predicado da linha 13 como verdadeiro e incluirá p_l em $updated_i$ (linha 15). Por conseguinte, p_l não será punido a partir de t' (linhas 16-18). (2) Se $p_j \notin STABLE$ é porque ele falha ou deixa o sistema em algum instante t'''. Em ambos os casos, p_j deixa de dar passos e portanto não pune p_l a partir de $t \geq t'$, $t \geq t'''$. $_{Lema1}\square$

Lema 2 *Todo processo que falha ou deixa o sistema em algum instante t, será permanentemente punido por qualquer processo estável a partir de algum $t' \geq t$.*

Prova. Uma vez que um processo p_j falha ou abandona o sistema em um instante t, ele para de executar TASK 1: ALIVE (linhas 4-6). Portanto, a partir de um instante $t' \geq t$, toda vez que um processo estável p_e executar TASK 2, encontrará $Alive[j] = last_alive_e[j]$ (linhas 12-13) e acabará punindo p_j. Uma vez que $p_e \in STABLE$, permanecerá executando a tarefa e punindo p_j. $_{Lema2}\square$

Lema 3 *Um processo p_j que falha ou deixa o sistema em um instante t jamais será eleito líder a partir de algum instante $t' \geq t$.*

Prova. (1) Pelo Lema 2, p_j será permanentemente punido por todo processo estável p_e a partir de algum $t'' \geq t$. Isso fará com que, a partir deste instante, os contadores $Punishments[e][j]$, $\forall p_e \in STABLE$, permaneçam crescendo monotonicamente (linhas 7-18). (2) Pelo Lema 1, existe um instante t''' após o qual algum $p_l \in STABLE$ jamais é punido. Portanto, a partir de t''' os contadores $Punishments[-][l]$ permanecem constantes. De (1) e (2) temos que a partir de um instante t', $t' \geq t''$ e $t' \geq t'''$,

$\sum_{\forall k} Punishments[k][j] > \sum_{\forall k} Punishments[k][l]$, o que impedirá que p_j seja escolhido líder (linhas 24-25). $Lema3\square$

Lema 4 *Se um processo $p_l \in STABLE$ tiver sido eleito líder em um instante t, um processo p_r que chega ao sistema em um instante $t' > t$ não demove o líder eleito.*

Prova. Quando um processo p_r entra no sistema, cada registrador $Punishments[k][r]$ é inicializado com o valor $Punishments[k][l] + 1$ (linha 3). O líder eleito p_l, por sua vez, é tal que $l = Min(\sum_{\forall k} Punishments[k][j], j)$ (linha 24). Portanto, uma vez que a soma dos contadores relativos a p_r, $\sum_{\forall k} Punishments[k][r]$, será maior que a soma dos contadores relativos a p_l, p_r jamais será atribuído a $leader_i$. $Lema4\square$

Teorema 1 *Existe um instante após o qual algum $p_l \in STABLE$ será eleito permanentemente líder.*

Prova. Segue direto dos Lemas 1, 2, 3, 4. $Teorema1\square$

5. Trabalhos Relacionados

A maioria das propostas de implementação de Ω em sistemas assíncronos apresentadas na literatura considera sistemas de passagem de mensagens. Nesse contexto, a busca por requisitos mínimos que possibilitem a implementação segue duas vertentes distintas: abordagens baseadas em tempo (*timer-based*) e abordagens baseadas em padrão de mensagem (*message-pattern*). A Tabela 1 resume as principais abordagens e suas características.

Abordagem	Comunicação	Π	Conectividade	Propriedade do Modelo
[Chandra et al. 1996]	pass. de mensagens	conhecido	grafo completo	todos os canais *eventually timely*
$\exists p$ *eventually accessible* [Aguilera et al. 2001]	pass. de mensagens	conhecido	grafo completo	$\exists p$: todos os canais bidirecionais de/para p são *eventually timely*
$\exists p$ que é $\Diamond f$-*source* [Aguilera et al. 2004]	pass. de mensagens	conhecido	grafo completo	$\exists p$: f canais de saída de p são *eventually timely*
$\exists p$ que é $\Diamond f$-*accessible* [Malkhi et al. 2005]	pass. de mensagens	conhecido	grafo completo	$\exists p, Q$: *round-trip* de p para $\forall q \in Q$, é *eventually timely*
$\exists p$ que é *eventually moving* f-*source* [Hutle et al. 2009]	pass. de mensagens	conhecido	grafo completo	$\exists p, Q$: canal de saída de p para $\forall q \in Q$, é *eventually timely*
[Jiménez et al. 2006]	pass. de mensagens	desconhecido dinâmico	nós corretos conectados	$\exists p$ correto: $\forall q$ correto, pode ser alcançado de p por canais *eventually timely*
[Fernández et al. 2006]	pass. de mensagens	desconhecido dinâmico	nós corretos conectados	$\exists p$ correto e $>$ (n-f)-$\alpha + 1$ processos corretos alcançáveis de p por canais *eventually timely*
[Guerraoui and Raynal 2006]	memória compartilhada	conhecido	-	sistema síncrono após um tempo
temporizadores "bem comportados" [Fernández et al. 2010]	memória compartilhada	conhecido	-	$\exists p$: cujos acessos à memória compartilhada são *eventually timely*
mecanismo *query-response* (*time-free*) [Mostefaoui et al. 2003]	pass. de mensagens	conhecido	grafo completo	$\exists t, p, Q$: a partir de t, $q \in Q$ recebe uma resposta de p à suas *queries* entre as $(n - f)$ primeiras.
mecanismo *query-response* (*time-free*) [Arantes et al. 2013]	pass. de mensagens	desconhecido dinâmico	grafo completo	$\exists t, p, Q$: a partir de t, $q \in Q$ recebe uma resposta de p à suas *queries* entre as α primeiras.

Table 1. Abordagens para Implementação de Ω

5.1. Abordagens Baseadas em Tempo

Estão relacionadas a suposições adicionais sobre limites para os passos de processos ou para transferência de mensagens. Geralmente, esses limites não precisam ser conhecidos

a priori além de que não precisam ser verificados durante todo tempo de execução do sistema. Os protocolos para sistemas assíncronos baseados nessa abordagem utilizam *time-outs* e diferem entre si, basicamente, na quantidade de conectividade e sincronia exigida para conversão. As nove primeiras entradas da Tabela 1 correspondem a essa abordagem.

Os primeiros estudos de detectores de líder consideravam uma rede de comunicação completamente conectada com todos os canais confiáveis e síncronos a partir de um determinado instante [Chandra et al. 1996]. A proposta de [Aguilera et al. 2001], considera que existem pelo menos um processo correto p e um instante t, tais que a partir de t todos os canais bidirecionais entre p e todos os outros processos são síncronos (*eventually timely*). Tal processo p é dito ser *eventually accessible*.

Em [Aguilera et al. 2004], a definição de *eventually timely link* é por eles apresentada juntamente com a de outras propriedades referentes a canais. *Integridade*: um processo p recebe uma mensagem m de q apenas uma vez e somente se p enviou m para q. Um canal é *confiável* (*reliable*) se ele satisfaz integridade além de que, se p envia uma mensagem m a um processo correto q, então q acaba por receber m. Se um canal satisfaz *justiça* (*fairness*), então, se um processo p envia um número infinito de mensagens de um determinado tipo, então o canal entrega um número infinito de mensagens daquele tipo. Um canal é *fair-lossy* se ele satisfaz integridade e *fairness*: se um processo p envia um número infinito de mensagens de um determinado tipo para q, e q é correto, então q recebe de p um número infinito de mensagens daquele tipo. Um canal é *timely* se satisfaz integridade e existe um δ tal que se p envia uma mensagem m a q no instante t, e q é correto, então q recebe m de p até o instante $t + \delta$.

Finalmente, um canal é *eventually timely* se satisfaz integridade e existem t e δ tais que se p envia uma mensagem m a q no instante $t' \geq t$, e q é correto, então q recebe m de p até o instante $t' + \delta$ (mensagens enviadas antes de t podem ser perdidas). Os autores então mostram que Ω pode ser implementado em sistemas com n processos dos quais até f podem falhar (*crash*), cujos canais sejam *fair-lossy*, desde que pelo menos um processo correto p possua f canais de saída que sejam *eventually timely*. Nesse caso, p é dito ser um $\Diamond f\text{-}source$. Adicionalmente, os autores mostram que se os canais forem confiáveis (*reliable*), é possível implementar um protocolo otimal, do ponto de vista da comunicação. Isso porque o limite inferior para o número de canais que precisam carregar mensagens para sempre em qualquer implementação de Ω livre de falhas é f.

Em [Malkhi et al. 2005], a estratégia adotada é diferente das anteriores; eles usam a noção de *eventually f-accessibility*. Em um sistema com n processos em que até $f \leq (n-1)/2$ podem falhar por parada e cujos canais são confiáveis, existe um limite superior δ para um *round-trip* de mensagens que é conhecido mas que não se mantém para todo canal em todo instante. Supõe-se que existem um instante t, um processo p e um conjunto Q de f processos, tais que, $\forall t' \geq t$, p troca mensagens com cada q dentro do limite δ, isto é, p é $\Diamond f$*-accessible*. No artigo é apresentado um protocolo baseado na suposição. Deve-se observar que o subconjunto de processos Q não é fixo, isto é, pode variar com o tempo.

Uma suposição ainda mais fraca é a proposta em [Hutle et al. 2009]: existe um processo correto p, a saber, um *eventually moving f-source*, tal que, após um tempo, cada mensagem que ele envia é recebida em tempo (*timely*) por um conjunto Q de f processos

que pode ser diferente em instantes distintos. A diferença básica da suposição anterior é que aqui só é exigido que os canais de saída de p sejam *eventually timely*. É assumido que um processo faltoso sempre recebe as mensagens em tempo. O trabalho mostra que a complexidade de comunicação para implementações baseadas nessa premissa é limitada inferiormente por (nt) e propõe três protocolos, dos quais um apresenta essa complexidade.

Como citado na Introdução, o projeto de um algoritmo para um sistema dinâmico precisa levar em conta a imprevisibilidade devida ao fato de que processos podem entrar e sair arbitrariamente do sistema. Obviamente, uma característica inerente a tais ambientes é a falta de conhecimento inicial que cada nó tem a respeito do conjunto de participantes. Uma abordagem tradicional em tais sistemas, quando a comunicação se dá por passagem de mensagem, é a utilização de primitivas de difusão a partir de cujas respostas os processos constroem seu conhecimento (*membership*) do sistema, ao passo em que também constroem o serviço de líder.

[Jiménez et al. 2006] propõem um protocolo para ambiente dinâmico com troca de mensagens, supondo que existe um limite inferior sobre o número de passos por unidade de tempo que um processo dá e que todos os processos corretos são conectados por canais *eventually timely*. Diferentemente do nosso, o protocolo apresentado é baseado em $timeouts$. [Fernández et al. 2006] apresentam dois protocolos. No primeiro eles consideram que os processos não conhecem o número de participantes n nem o limite para o número de falhas f. Como na nossa abordagem, cada processo só conhece sua própria identidade e um limite inferior α sobre o número de processos corretos. Essa constante α abstrai todos os pares (n, f) tradicionalmente consideradas como conhecidas, tal que $n - t \geq \alpha$. Dessa forma, um protocolo baseado em α, funciona para qualquer par (n, f). Diferentemente da nossa proposta, entretanto, os processos se comunicam por troca de mensagens e a conversão do algoritmo, baseado em *timeouts*, exige a existência de alguns canais *eventually timely*.

No que diz respeito ao paradigma de comunicação por memória compartilhada, muito poucas são as propostas encontradas na literatura para a implementação de Ω, a maioria delas para sistemas estáticos. [Guerraoui and Raynal 2006] implementam um protocolo com registradores regulares compartilhados, mas diferentemente do nosso trabalho, consideram um sistema estático, síncrono após um tempo. [Fernández et al. 2007, Fernández et al. 2010], assim como nós, consideram um sistema assíncrono com registradores atômicos compartilhados. Ao contrário da nossa proposta, entretanto, o ambiente é estático e eles fazem suposições adicionais sobre o sincronismo do sistema para atingir a conversão do protocolo, o qual é baseado em *timeouts*.

Em um sistema de memória compartilhada cujos participantes são desconhecidos, um processo que entra na computação, naturalmente, precisa inteirar-se do estado do sistema antes de começar a participar efetivamente. [Baldoni et al. 2009] propõem uma abstração de registrador regular compartilhado no topo de um sistema dinâmico de passagem de mensagens. Quando um processo p_i deseja entrar, invoca uma operação $join()$, que lhe permite obter um estado consistente do registrador. No início da operação, p_i envia um *broadcast* com sua identidade informando aos demais processos sobre sua chegada. Os processos p_j respondem, informando sua identidade e o valor local de seu registrador. p_i então processa os dados recebidos, definindo um valor consistente para o registrador.

Ao terminar $join()$, p_i está apto a realizar operações $read$ e $write$ sobre o objeto compartilhado.

De modo semelhante, o nosso protocolo envolve a invocação de uma operação $join()$ que se encarrega de criar registradores devidamente inicializados, fornecer ao recém-chegado, p_r, uma cópia consistente de cada registrador compartilhado, e incluí-lo na computação. A implementação da operação $join()$, entretanto, está fora do escopo do nosso trabalho. O algoritmo considera que ao concluir a operação, p_r estará apto a trocar informações com os demais participantes através da memória compartilhada e vice-e-versa. Além disso, o estado inicial de p_r será tal que impeça a demoção equivocada de um líder estável.

Outras implementações de Ω disponíveis na literatura, assim como a nossa, não consideram suposições adicionais relativas à sincronia de processos ou canais de comunicação. Ao invés disso, baseiam-se em um padrão de troca de mensagens. Até onde sabemos, não existe uma implementação de Ω para sistemas assíncronos de memória compartilhada estendidos com suposições adicionais baseadas em tempo.

5.2. Abordagens Baseadas em Padrão de Mensagens

A fim de possibilitar a implementação de detectores de falhas em um sistema assíncrono, [Mostefaoui et al. 2003] introduziram uma nova abordagem. Eles se baseiam em um mecanismo *query-response*, assumindo que a resposta de algum processo p_i às últimas *queries* requisitadas por determinados processos p_j está sempre entre as primeiras $n - f$ respostas recebidas por cada p_j, onde n é o número total de processos no sistema e f um limite superior para o número de falhas. Eles mostram que se essa suposição é satisfeita a partir de um instante $t \geq 0$, um detector $\Diamond S$ pode ser implementado. A intuição por trás da suposição é a de que, mesmo que o sistema não exiba propriedades síncronas, ele pode apresentar alguma regularidade de comportamento que se traduza em uma *sincronia lógica* capaz de contornar a impossibilidade de conversão do algoritmo [Mostefaoui et al. 2006].

Considerando uma suposição semelhante, [Arantes et al. 2013] propõem um modelo para implementação de Ω em um sistema dinâmico de passagem de mensagens que leva em conta a mobilidade dos nós durante a computação. Para garantir a corretude do protocolo, eles consideram uma versão da suposição baseada em padrão de mensagens adaptada a sistemas dinâmicos segundo a qual os processos envolvidos devem pertencer a um conjunto de nós *estáveis*, isto é, processos que, uma vez que se ligaram ao sistema no instante t, permanecem para sempre. Além disso, eles utilizam o limite inferior α sobre o número de processos corretos para controlar o número de respostas consideradas entre as primeiras. No protocolo aqui apresentado, introduzimos uma suposição semelhante, isto é, o modelo considera uma noção de sincronia lógica, porém com relação aos acessos à memória compartilhada.

6. Considerações Finais

Os resultados aqui reportados trazem uma primeira contribuição ao estudo do problema de líder em ambientes dinâmicos de memória compartilhada. Introduzimos uma propriedade comportamental que reflete um padrão relativo aos acessos feitos à memória compartilhada. No modelo considerado, a memória é constituída de registradores atômicos com-

partilhados *1-escritor-N-leitores*, tais que o registrador $R[i]$ é escrito pelo processo proprietário p_i e lido por qualquer processo p_j participante no sistema. Dessa maneira, cada processo p_i pode manifestar seu progresso incrementando $R[i]$ periodicamente. Cada p_j pode então conferir se cada p_i permanece vivo, observando o crescimento de $R[i]$.

A propriedade que propomos considera esse acompanhamento que cada p_j faz de cada $R[i]$. Informalmente, supomos que existe um processo *estável* p_l e um instante t, após o qual, as escritas de p_l em $R[l]$ estão sempre entre as primeiras α atualizações percebidas entre todos os registradores compartilhados, do ponto de vista de qualquer processo *estável* no sistema. A noção de *estável* refere-se a um processo que entra no sistema e permanece vivo (não falha ou deixa o sistema) durante toda computação. A existência de um conjunto de processos estáveis em um sistema dinâmico é necessária para que se possa realizar alguma computação útil. O parâmetro α representa o limite inferior para a cardinalidade deste conjunto.

Em resumo, as principais contribuições deste artigo foram:

- A definição de um modelo de sistema dinâmico assíncrono de memória compartilhada sujeito a uma propriedade baseada em um padrão de acesso à memória (*time-free*) que permite a implementação de um serviço de líder;
- Um protocolo que implementa um serviço de líder da classe Ω adequado ao modelo proposto.

Como trabalho futuro pretende-se avaliar a possibilidade de uso de registradores regulares ao invés de atômicos, já que os primeiros são mais fracos. Outra possibilidade é diminuir a complexidade do protocolo, de tal maneira que somente o líder tenha que escrever para sempre em seus registradores, enquanto os demais processos precisarão apenas ler.

References

Aguilera, M., Delporte-Gallet, C., Fauconnier, H., and Toueg, S. (2004). Communication-efficient leader election and consensus with limited link synchrony. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, pages 328–337, New York, NY, USA. ACM.

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

Aguilera, M. K., Delporte-Gallet, C., Fauconnier, H., and Toueg, S. (2001). Stable leader election. In Welch, J., editor, *Distributed Computing*, volume 2180 of *Lecture Notes in Computer Science*, pages 108–122. Springer Berlin Heidelberg.

Aguilera, M. K., Englert, B., and Gafni, E. (2003). On using network attached disks as shared memory. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, PODC '03, pages 315–324, New York, NY, USA. ACM.

Arantes, L., Greve, F., Sens, P., and Simon, V. (2013). Eventual leader election in envolving mobile networks. In *Proceedings of the 17th International Conference on Principles of Distributed Systems*, OPODIS '13, Berlin, Heidelberg. Springer-Verlag.

Baldoni, R., Bonomi, S., Kermarrec, A.-M., and Raynal, M. (2009). Implementing a register in a dynamic distributed system. In *Distributed Computing Systems, 2009. ICDCS '09. 29th IEEE International Conference on*, pages 639–647.

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

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

Fernández, A., Jiménez, E., and Raynal, M. (2006). Eventual leader election with weak assumptions on initial knowledge, communication reliability, and synchrony. In *In DSN*, pages 166–175.

Fernández, A., Jiménez, E., and Raynal, M. (2007). Electing an eventual leader in an asynchronous shared memory system. In *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*, pages 399–408.

Fernández, A., Jiménez, E., Raynal, M., and Trédan, G. (2010). A timing assumption and two t-resilient protocols for implementing an eventual leader service inÂ asynchronous shared memory systems. *Algorithmica*, 56(4):550–576.

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

Gafni, E. and Lamport, L. (2003). Disk paxos. *Distrib. Comput.*, 16(1):1–20.

Guerraoui, R. and Raynal, M. (2003). The information structure of indulgent consensus. *IEEE Transactions on Computers*, 53:2004.

Guerraoui, R. and Raynal, M. (2006). A leader election protocol for eventually synchronous shared memory systems. In *Software Technologies for Future Embedded and Ubiquitous Systems, 2006 and the 2006 Second International Workshop on Collaborative Computing, Integration, and Assurance. SEUS 2006/WCCIA 2006. The Fourth IEEE Workshop on*, pages 6 pp.–.

Guerraoui, R. and Raynal, M. (2007). The alpha of indulgent consensus. *Comput. J.*, 50(1):53–67.

Herlihy, M. P. and Wing, J. M. (1990). Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492.

Hutle, M., Malkhi, D., Schmid, U., and Zhou, L. (2009). Chasing the weakest system model for implementing ω and consensus. *IEEE Transactions on Dependable and Secure Computing*, 6(4):269–281.

Jiménez, E., Arévalo, S., and Fernández, A. (2006). Implementing unreliable failure detectors with unknown membership. *Information Processing Letters*, 100(2):60 – 63.

Malkhi, D., Oprea, F., and Zhou, L. (2005). ω meets paxos: Leader election and stability without eventual timely links. In *Proceedings of the 19th International Conference on Distributed Computing*, DISC'05, pages 199–213, Berlin, Heidelberg. Springer-Verlag.

Mostefaoui, A., Mourgaya, E., and Raynal, M. (2003). M.: Asynchronous implementation of failure detectors. In *In: Proc. International IEEE Conference on Dependable Systems and Networks (DSN'03*, pages 351–360.

Mostefaoui, A., Mourgaya, E., Raynal, M., and Travers, C. (2006). A time-free assumption to implement eventual leadership. *Parallel Processing Letters*, 16(02):189–207.

32º Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos

Florianópolis - SC

XV Workshop de Testes e Tolerância a Falhas (WTF 2014)

Sessão Técnica 3

Tolerância a Falhas, Disponibilidade, Segurança e DTN

Uma Avaliação de Abordagens de Distribuição para Gerenciamento de Redes Tolerantes a Atrasos e Desconexões*

Ewerton M. Salvador[1], Vinícius Mota[1], Virgil Almeida[1], Daniel F. Macedo[1], José Marcos Nogueira[1], Jéferson C. Nobre[2], Pedro A. P. R. Duarte[2], Lisandro Z. Granville[2], Liane M. R. Tarouco[2]

[1] Universidade Federal de Minas Gerais (UFMG)
Departamento de Ciência da Computação
Belo Horizonte – MG – Brasil

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

{ewerton,vfsmota,virgil,damacedo,jmarcos}@dcc.ufmg.br

{jcnobre,paprduarte,granville,liane}@inf.ufrgs.br

Abstract. *The absence of some key TCP/IP assumptions in Delay-Tolerant Networks (DTNs) makes the employment of conventional management approaches more difficult. This paper evaluates the strengths and weaknesses of the traditional centralized, hierarchical, and distributed management approaches being used in a DTN environment. A DTN simulator was used for evaluating two metrics regarding the use of these management approaches: the success poll rate and the latency of the monitoring operation. The results achieved in this work have potential for guiding the development of new solutions for managing Delay-Tolerant Networks and improving its fault tolerance.*

Resumo. *A ausência de pressuposições básicas do TCP/IP em Redes Tolerantes a Atrasos e Desconexões (DTNs) faz com que o emprego de abordagens convencionais de gerenciamento seja mais difícil. Este artigo avalia as forças e fraquezas das tradicionais abordagens de gerenciamento centralizada, hierárquica e distribuída, quando empregadas em um ambiente DTN. Um simulador DTN foi usado para avaliar duas métricas relacionadas ao uso das abordagens de gerenciamento: a taxa de sucesso de operações de polling e a latência da operação de monitoramento. Os resultados alcançados no trabalho possuem potencial para guiar o desenvolvimento de novas soluções para o gerenciamento de DTNs e melhorar sua tolerância a falhas.*

1. Introdução

Redes Tolerantes a Atrasos e Desconexões (*Delay-Tolerant Networks*, ou DTNs) foram desenvolvidas para permitir comunicação em ambientes com desconexões frequentes e longos atrasos de transmissão [Cerf et al. 2007]. Apesar da tecnologia DTN ter sido inicialmente criada para comunicações interplanetárias, essas redes também podem beneficiar aplicações terrestres, especialmente em cenários onde a comunicação é assíncrona

*Os autores gostariam de agradecer ao CNPq, à CAPES e à FAPEMIG pelo suporte a este trabalho.

e sujeita a grandes atrasos de entrega [Ivansic 2009]. Esse tipo de aplicação é comum em redes esparsas e/ou altamente móveis, ou em redes onde os links possuem altas taxas de erro. Como exemplos de DTNs aplicadas em cenários terrestres, pode-se citar redes para entrega de mensagens em áreas subdesenvolvidas ou rurais[wiz 2010], comunicação militar[Parikh and Durst 2005], ou para a coordenação de equipes de emergência trabalhando na mitigação de desastres[Fall 2003].

Assim como em qualquer rede de computadores, DTNs precisam ser gerenciadas a fim de identificarem e tratarem falhas de forma eficiente e eficaz. Contudo, as abordagens de gerenciamento existentes podem não ser adequadas para DTNs porque elas geralmente assumem que o gerente se comunica com os elementos gerenciados usando caminhos fim-a-fim estáveis. Portanto, se faz necessária a observação dessas abordagens de gerenciamento em um ambiente de Redes Tolerantes a Atrasos e Desconexões, de modo a se melhor compreender até onde essas abordagens precisam ser adaptadas para DTNs ou se elas precisam ser substituídas por abordagens completamente novas e específicas para DTNs. Por exemplo, não se pode ter certeza sobre o atraso que ocorrerá para que alarmes ou comandos de gerenciamento cheguem aos seus destinos, ou mesmo se eles chegarão aos seus destinos nas DTNs. Assim, o gerenciamento de redes DTN (*Delay-Tolerant Network Management*, ou DTNM) é um problema de pesquisa em aberto[Birrane and Cole].

A eficácia das abordagens de distribuição do gerenciamento depende dos tipos de tarefas de gerenciamento a serem realizadas na rede, assim como de seus requisitos de QoS. Tarefas de gerenciamento de tempo real, tais como o controle de parâmetros das camadas MAC e PHY, provavelmente irão exigir abordagens distribuídas. Contudo, a quantidade de mensagens de gerenciamento e o custo de se ter vários gerentes penalizam o uso de recursos dessa abordagem de distribuição. Outras abordagens poderiam ser empregadas, tais como as hierárquicas, onde os nós são organizados em um ou mais níveis de *clusters*. Finalmente, pode-se empregar a abordagem centralizada, onde um único nó gerencia a rede inteira. Obviamente essas abordagens devem funcionar apenas para operações de gerenciamento que são insensíveis a atrasos e altas taxas de perda de pacotes.

Para se melhor compreender quais abordagens de distribuição são mais adequadas para cada tarefa de DTNM, suas forças e fraquezas precisam ser investigadas. Seus desempenhos dependem das características da rede, tais como os padrões de mobilidade dos nós, o tamanho dos buffers de mensagens, as tarefas de gerenciamento a serem executadas, e a estratégia de roteamento.

Este artigo investiga através de simulações o comportamento de diferentes abordagens de distribuição do gerenciamento quando aplicadas ao gerenciamento de DTNs. Foram investigados os usos das abordagens centralizada, hierárquica e distribuída aplicadas em um cenário de redes de emergência. O objetivo dessa pesquisa é identificar quais classes de tarefas de gerenciamento são factíveis em cada abordagem de distribuição. Esse estudo é um primeiro passo em direção a uma melhor compreensão do problema de gerenciamento DTN, uma vez que ele proporciona resultados de desempenho quantitativos para guiar a construção de novas soluções para DTNM. Uma vez que esses comportamentos sejam compreendidos para cada abordagem de gerenciamento em um ambiente DTN, poderão ser desenvolvidos elementos de arquiteturas de gerenciamento para funções DTN

específicas, objetivando o estabelecimento do arcabouço básico necessário para se ter mecanismos de tolerância a falhas.

O restante desse artigo está organizado como se segue. A Seção 2 apresenta a contextualização necessária para a compreensão do restante do trabalho. A metodologia da simulação é fornecida na Seção 3. A Seção 4 apresenta e discute os resultados das simulações. Finalmente, conclusões e trabalhos futuros são descritos na Seção 5.

2. Contextualização

Recentemente, o conceito de Redes Tolerantes a Atrasos e Desconexões tem recebido bastante atenção tanto da indústria quanto da academia. Nessas redes, algumas pressuposições assumidas na arquitetura TCP/IP são relaxadas, tais como a existência de um caminho fim-a-fim entre os nós fonte e destino, um tempo de *round-trip* (RTT) não excessivo entre qualquer par de nós, e uma baixa probabilidade de descarte de pacotes[Cerf et al. 2007]. As origens da pesquisa sobre DTN estão nos esforços conduzidos pela NASA Jet Propulsion Laboratory para estabelecer uma Internet Interplanetária (*InterPlanetary interNet*, ou IPN). Contudo, é possível se encontrar oportunidades para a utilização de recursos oriundos das DTNs em cenários terrestres, tais como acesso a internet para vilas remotas, redes veiculares, comunicação em campo de batalha e redes de emergência. É importante ressaltar que ainda são poucos os trabalhos existentes na literatura que abordam o gerenciamento de DTNs (DTNM).

Quando uma determinada região enfrenta uma situação de desastre é comum que sua infraestrutura de telecomunicação seja perdida. Objetivando a orquestração dos esforços para mitigar a emergência, a comunicação e a coordenação entre os times de resgate são essenciais, uma vez que um trabalho rápido e bem coordenado ajuda a salvar vidas e a controlar perdas econômicas. O padrão atual para comunicação em cenários de emergência é o uso de rádio de voz. Apesar do baixo custo e da simplicidade do emprego dessa solução, o uso de rádios possui uma série de limitações, tais como a incapacidade para transmitir informações gráficas (e.g., fotos e mapas), não permite o armazenamento e organização de informações históricas, e apresenta baixa velocidade de transmissão de dados [Calarco and Casoni 2011]. Uma rede de emergência é uma rede ad hoc montada em uma região que sofreu um evento desastroso, de modo a melhorar a diversidade de suportes informacionais (e.g., texto, mapas, imagens, vídeos e voz) que possam ser usados na tarefa de comunicação em uma operação de mitigação do desastre. Alguns fatores podem quebrar a conectividade de uma rede nesse tipo de cenário, tais como obstáculos naturais, destroços e o raio de comunicação limitado dos dispositivos móveis. Devido ao fato de desconexões serem um problema comum nesse tipo de rede, o uso da arquitetura DTN permite o uso de dispositivos computacionais móveis em ambientes de desastre. É possível encontrar na literatura um número de trabalhos apresentado soluções baseadas em DTN para esse tipo de ambiente [Jiang et al. 2011, Dong et al. 2009].

O gerenciamento de redes DTN pode ser implementado por diferentes modelos de produção de dados de gerenciamento. Algumas investigações preliminares indicam que esse gerenciamento deve ser específico a sub-redes, usando um protocolo de gerenciamento específico para sub-redes [Birrane and Cole]. Por exemplo, em ilhas de baixa latência, tarefas de gerenciamento podem ser desempenhadas usando protocolos que não sejam DTN, tais como SNMP ou NETCONF. Nesse contexto, dados de gerenciamento

podem ser posteriormente armazenados e processados em um nó intermediário (*i.e.*, um *peer* de gerenciamento) antes de ser enviado adiante. Esse procedimento pode ser visto como um modelo de "telemetria", onde alguns dados de gerenciamento processados podem ser enviados em intervalos regulares (*e.g.*, a média de um objeto de uma *Management Information Base* - MIB). Dados de gerenciamento também podem ser enviados em resposta a uma requisição (*e.g.*, disponibilidade de um sistema de monitoramento através de uma ferramenta semelhante a um ping). Contudo, nenhum tipo de validação foi publicada até o momento.

Nesse contexto, soluções de gerenciamento DTN podem ser concretizadas por diferentes abordagens de gerenciamento de redes, dependendo das características almejadas pelas diferentes soluções. Ao longo dos anos, houve algumas tentativas de organizar as abordagens propostas em taxonomias, especialmente no que diz respeito aos problemas de distribuição.

As principais taxonomias para gerenciamento de redes distribuídos na literatura foram propostas por Martin-Flatin *et al.* [Martin-Flatin et al. 1999] e Schoenwaelder *et al.* [Schönwalder et al. 2000]. Martin-Flatin *et al.* [Martin-Flatin et al. 1999] propôs uma taxonomia simples e uma melhorada. A primeira se baseia em critérios de organização e divide o gerenciamento de redes em centralizado, hierárquico fracamente distribuído, hierárquico fortemente distribuído, e cooperativo fortemente distribuído. Schoenwaelder *et al.* [Schönwalder et al. 2000] introduz uma taxonomia muito similar a versão simples de Martin-Flatin. Contudo, os autores categorizaram as abordagens de gerenciamento de redes considerando apenas o número de gerentes e agentes. Essa taxonomia possui quatro categorias de gerenciamento: centralizado, fracamente distribuído, fortemente distribuído e cooperativo.

Neste trabalho é utilizada uma taxonomia similar a versão simples de Martin-Flatin, onde se divide as abordagens de gerenciamento em centralizada, onde existe um único nó que gerencia todos os agentes da rede; distribuído, onde cada nó da rede atua tanto como gerente quanto como agente; e hierárquica, onde vários nós organizados hierarquicamente desempenham a tarefa de gerenciar os agentes da rede através da coordenação de suas ações, frequentemente através da delegação de tarefas de um para o outro. Acredita-se que essa organização permite uma boa investigação inicial da distribuição de nós de gerenciamento em uma rede DTN.

3. Metodologia

A investigação apresentada neste artigo sobre o uso de abordagens convencionais de distribuição do gerenciamento em DTN foi realizada através de simulações, uma vez que essa prática proporciona informação significativa sobre os possíveis problemas que podem ocorrer sem que se tenha o custo de se implementar um *testbed* com dispositivos reais. O simulador "Opportunistic Network Environment" (ONE) [Keränen et al. 2009] foi escolhido por já ser bem estabelecido em meio a comunidade que pesquisa redes DTN.

3.1. Os Cenários

O gerenciamento das Redes Tolerantes a Atrasos e Desconexões será influenciado pelas aplicações que irão ser atendidas pela rede, uma vez que várias decisões chave acerca da configuração da DTN são baseadas em aplicações, tais como: algoritmo de roteamento,

mecanismos de segurança, protocolo de transporte da rede subjacente, etc. Por conta disso, a avaliação descrita neste artigo precisa considerar um tipo de aplicação para guiar o preparo do experimento com simulação (e.g., padrões de mobilidade, requisitos de desempenho das tarefas de gerenciamento, etc.). Portanto, foi considerada uma rede de emergência para mitigação de desastre como nosso principal cenário de simulação. Essas são redes que podem ser prontamente instaladas em situações de desastres, onde a infraestrutura de comunicação local é perdida, de modo a apoiar a comunicação e coordenação dos times de resgate.

Cada uma das abordagens de distribuição (centralizada, distribuída e hierárquica) pode ser implementada em redes reais por um diferente número de arquiteturas de gerenciamento. Para esse estudo em específico, as simulações se guiam por tecnologias baseadas no Simple Network Management Protocol (SNMP) [Case et al. 1990], tais como o SNMP em si e o SCRIPT-MIB [Levi and Schoenwaelder 2001]. A razão para a escolha dessas tecnologias é o fato do SNMP ainda ser o padrão *de facto* para o gerenciamento de redes de computadores. Contudo, é importante ressaltar que foca-se nesse estudo apenas a forma como as entidades de cada uma das abordagens se comunicam entre si e o impacto das DTNs nessas comunicações. Outras características, tais como desempenho, estruturas de dados e segurança não são consideradas neste artigo. A única operação de gerenciamento que irá ser avaliada no escopo deste trabalho é **monitoramento**, uma vez que ela é uma das funções mais fundamentais em sistemas de gerenciamento e de tolerância a falhas.

3.2. Modelos de Mobilidade

Outro aspecto importante da simulação DTN é a escolha do modelo de mobilidade, a qual precisa estar de acordo com as aplicações que irão ser instanciadas na rede. Nas simulações realizadas neste trabalho, foram escolhidos dois modelos de mobilidade: *Random Way Point* (RWP) [Rojas et al. 2005] e *Reference Point Group Mobility* (RPGM) [Hong et al. 1999].

O RWP é um modelo de mobilidade simples onde um nó da rede escolhe um destino aleatório, se dirige para esse destino usando uma velocidade aleatória, e permanece parado naquele ponto for um período de tempo aleatório. Esse modelo de mobilidade é frequentemente usado em experimentos onde o objetivo é prover análises genéricas sobre o comportamento de uma tecnologia de rede que está sendo simulada.

O modelo RPGM é similar ao modelo *Random Way Point* no que diz respeito aos nós escolherem destinos, velocidades e pausas aleatórias. Contudo, no modelo RPGM os nós se movem em grupos, onde cada grupo possui um "centro" lógico, o qual define o movimento do grupo inteiro. Os nós estão geralmente uniformemente distribuídos ao redor desse "centro" lógico, e eles também podem se mover independentemente do movimento do grupo, dentro de uma distância máxima do centro lógico. Esse modelo de mobilidade é comumente utilizado em simulações de redes ad hoc onde a formação de grupos ocorre, como redes de emergência e comunicação em campo de batalha [Hong et al. 1999].

3.3. Métricas da Simulação

Uma vez que o interesse desse estudo é avaliar os aspectos de comunicação das abordagens de gerenciamento selecionadas, de forma a determinar até onde elas são adequadas

ou não para ambientes DTN, foram escolhidas duas métricas para a análise dos resultados das simulações: a taxa de sucesso de *polling*, que no escopo deste trabalho é o número de mensagens de resposta que os gerentes da rede receberam como resultado de mensagens de requisição de gerenciamento enviadas anteriormente, e a latência da operação de monitoramento. Aqui se considera a latência como o intervalo de tempo entre a criação da mensagem de requisição de monitoramento e a entrega de sua respectiva resposta. À medida que essas métricas forem sendo analisadas, será possível determinar quais aspectos das abordagens de gerenciamento estudadas funcionam corretamente num ambiente DTN, e quais irão apresentar algum tipo de problema ou deficiência, como será detalhado na próxima sessão.

3.4. Principais Parâmetros da Simulação

Neste trabalho, uma rede de emergência empregando abordagens de gerenciamento de rede convencionais (centralizada, hierárquica e distribuída) é simulada em um ambiente DTN. O número de nós em cada cenário variou de 20 a 61 (conforme detalhado na Seção 4), divididos em dois, quatro ou seis grupos de nós. Apesar das redes serem relativamente pequenas, esses números são suficientes para demonstrar os aspectos de comunicação que estão sendo estudados. Também foi feito uso de um algoritmo de roteamento ótimo hipotético. Nesse algoritmo, a largura de banda é infinita, todas as mensagens são entregues aos seus destinos sempre que isso for possível usando a menor quantidade de saltos, e o tempo de processamento da mensagem em cada salto é insignificante. Isso reduz o número de variáveis no nosso estudo, uma vez que não existe interesse em qual algoritmo de roteamento DTN irá se adequar melhor a um ambiente de gerenciamento DTN específico. Os cenários foram configurados para gerar requisições apenas nas três primeiras horas da simulação, de forma a proporcionar algum tempo para que todas as mensagens sejam entregues aos seus destinos durante a última hora de simulação. Os principais parâmetros da simulação estão descritos na Tabela 1. Cada um dos cenários simulados foi executado 15 vezes, e os resultados foram gerados com um intervalo de confiança de 95%.

Tabela 1. Principais parâmetros da simulação

Parâmetro	Valor
Área de Simulação	4000m X 4000m
Tempo Total de Simulação	4 horas
Alcance da Transmissão	250m (WiFi)
Modelos de Mobilidade	Random Way Point e RPGM
Velocidade dos Nós	0.5 m/s a 1.5 m/s
Tempo de Espera	0s a 120s

4. Resultados

Essa seção irá apresentar e discutir os resultados das simulações. Ela é dividida em subseções, uma para cada abordagem de gerenciamento. Cada simulação descrita foi executada 15 vezes, e os resultados apresentados nas próximas subseções possuem intervalo de confiança de 95%.

4.1. Abordagem Centralizada

A implementação de uma aplicação de gerenciamento centralizado de rede utilizada neste trabalho foi fortemente inspirada no protocolo SNMP. Um nó atua como o gerente da rede, monitorando todos os outros nós da rede. Os nós monitorados são os agentes da rede, os quais são responsáveis por responder às requisições do gerente com dados sobre o estado atual do nó.

Em cada um dos cenários estudados, existe um gerente estático localizado no centro da área da simulação (x=2000, y=2000). Além disso, existem 20, 40 e 60 agentes, divididos em grupos de 10 nós. É importante ressaltar que o gerente não pertence a nenhum desses grupos. O gerente envia uma mensagem *get-request* para todos os agentes a cada 5 minutos, requisitando dados de um objeto específico da MIB do agente, como por exemplo o nível de energia remanescente na bateria do dispositivo. Assim que a mensagem de requisição chega no agente, esse agente responde com o envio de uma mensagem *get-response* contendo os dados requisitados pelo gerente. É importante notar que o agente pode não conseguir se comunicar com o gerente da rede por longos períodos de tempo, e quando esse agente estiver finalmente conectado, ele pode receber múltiplas requisições ao mesmo tempo. De modo geral, esse comportamento não acomoda atualizações periódicas.

No contexto desse trabalho acredita-se que a dificuldade mais importante em se utilizar a abordagem centralizada em DTNs é que existe um único nó responsável por originar todas as requisições e receber todas as respostas da rede inteira. Esse comportamento dificilmente é desejável em DTNs, uma vez que as conexões entre o gerente e os agentes são esporádicas. Nos próximos parágrafos será demonstrado como essa abordagem desempenha na rede de emergência simulada neste trabalho, usando dois modelos de mobilidade: RWP e RPGM.

`Modelo RWP`: A distribuição uniforme de nós na área de simulação, causada pelo uso do modelo RWP, contribui para a alta probabilidade de uma mensagem ser entregue com sucesso ao seu destino através do uso das técnicas de roteamento DTN *multi-hop*.

A análise da Figura 1 (imagem esquerda) mostra que o gerente foi bem sucedido em receber a maioria das respostas dos agentes. Além disso, a Figura 1 mostra que a taxa de sucesso do processo de *polling* na abordagem centralizada aumenta à medida em que a rede vai ser tornando mais densa. O cenário com 2 grupos de agentes apresentou uma taxa de sucesso nas requisições de *polling* de 83,15%. Com 4 grupos foi observada uma taxa de sucesso de 93.57% e com 6 grupos essa taxa foi de 98,01%. A lógica por trás desses resultados é simples: com mais nós espalhados na área de simulação, existe maior possibilidade da formação de um "caminho" ao longo do tempo que permite que uma mensagem viagem da sua origem ao seu destino. A densidade da rede também diminui a média de atraso registrada entre as emissões de uma mensagem *get-request* e a recepção de uma *get-response*. Com dois grupos foi registrado um atraso médio de 4254 segundos, com quatro grupos essa média foi de 2927 segundos e com 6 grupos ela foi de 2356 segundos, conforme apresentado na Figura 1 (imagem direita).

`Modelo RPGM`: Quando o modelo RPGM é empregado, os grupos de nós formam "ilhas", devido a proximidade física entre os nós dentro do grupo, deixando uma quantidade considerável de espaços vazios na área de simulação. Esse modo de posicio-

nar os nós torna a disseminação de mensagens dentro do grupo mais fácil, uma vez que a quantidade de contatos entre os nós do grupo é consideravelmente alta. Contudo, um "caminho" entre o gerente e um agente é mais difícil de ser formado, uma vez que as oportunidades de contato surgem apenas quando um grupo passa pelo local onde o gerente está posicionado, ou quando um grupo encontra outro grupo de modo a repassar uma cópia da mensagem na esperança de que esse último possa entrega-la ao seu destino final. Esse fato é ilustrado na Figura 1 (imagem esquerda), a qual mostra a taxa de sucesso das operações de requisição-resposta entre o gerente e os agentes da rede para o modelo RPGM em comparação com o modelo RWP. Essa taxa de sucesso é 65,48% no cenário com dois grupos, 75,45% com quatro grupos e 79,80% com seis grupos.

O atraso médio nas comunicações entre o gerente e um agente na rede também apresentou um desempenho baixo no modelo RPGM, quando comparado com o modelo RWP. O cenário com dois grupos de agentes é o único que apresentou um desempenho maior no modelo RPGM, com um atraso médio de 3182 segundos. Os outros dois casos apresentaram desempenho mais fraco do que os das simulações com o RWP, tendo o cenário com quatro grupos apresentado um atraso médio na comunicação requisição-resposta de 3411 segundos e o cenário com 6 grupos apresentado 3132 segundos de atraso médio. Também se pode observar que o impacto da densidade da rede no nosso estudo foi mais fraco nos cenários com o modelo RPGM em comparação com os cenário com o modelo RWP.

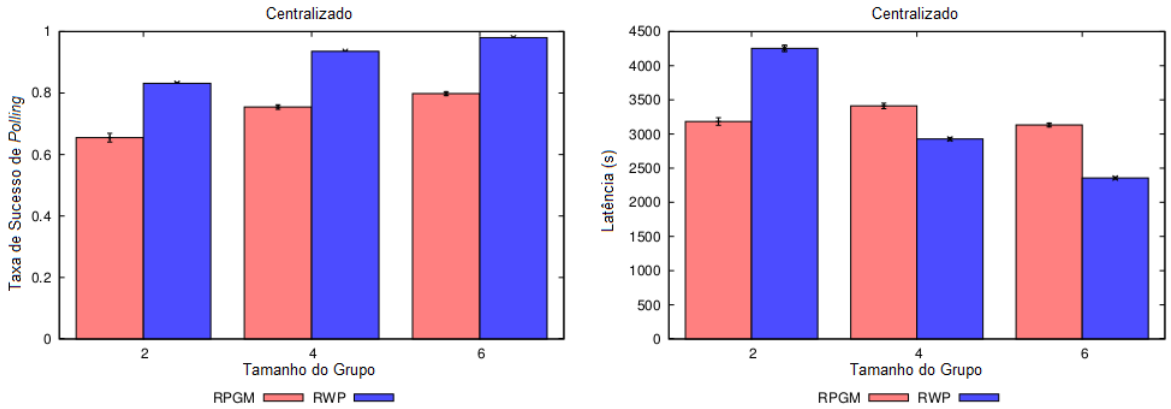


Figura 1. Análise das métricas na abordagem centralizada

4.2. Abordagem Hierárquica

Foi implementada uma aplicação de gerenciamento consistente com a abordagem hierárquica inspirada no Script-MIB [Levi and Schoenwaelder 2001], o qual é uma arquitetura de gerenciamento hierárquica desenvolvida para redes convencionais. Nessa aplicação, existe um gerente de nível mais alto (*top-level manager*, ou TLM) responsável por reunir informações acerca de toda a rede DTN, assim como delegar tarefas para gerentes secundários, conhecidos como gerentes de nível intermediário (*mid-level managers*, ou MLMs). Em nossos cenários simulados que usaram a abordagem hierárquica, os grupos de nós são formados por um MLM e nove agentes. Os agentes são responsáveis por responder as requisições enviadas pelos MLMs, os quais, por sua vez, são responsáveis por monitorar seus respectivos grupos.

Assim como ocorreu com as simulações usando a abordagem centralizada de gerenciamento, o TLM ficou localizado na área central da simulação (x=2000, y=2000), e ali permaneceu estacionário. O TLM não pertence a nenhum grupo de nós. O número de grupos foi variado em dois, quatro e seis, cada grupo sendo composto por 10 nós. O nó TLM enviou uma mensagem *MLM-request* para todos os MLMs da DTN a cada cinco minutos. Essa mensagem informa a um MLM que todos os agentes em seu grupo devem reportar o estado de um objeto específico da sua MIB. Em nosso exemplo, a energia restante nas baterias dos dispositivos foi sondada, assim como foi feito nas simulações usando a abordagem centralizada. O MLM executa sua ação enviando uma mensagem *get-request* para os agentes do seu grupo. Esses agentes respondem ao MLM com uma mensagem *get-response* contendo os dados requisitados. Ambos MLMs e os agentes podem receber múltiplas requisições de seus respectivos gerentes por conta das desconexões que ocorrem na DTN, assim como aconteceu quando foi utilizada a abordagem centralizada. Após 3 horas de simulação, o TLM parou de enviar mensagens *MLM-request* para dar às mensagens restantes na rede tempo para alcançarem seus destinos finais.

O principal interesse na simulação dessa aplicação é descobrir se a adição de um conjunto de gerentes organizados hierarquicamente pode melhorar o desempenho geral da tarefa de gerenciamento em uma DTN. A seguir será mostrado como essa abordagem desempenha nos cenários usando os modelos de mobilidade RWP e RPGM.

`Modelo RWP`: Nas simulações que empregaram o modelo RWP todos os nós DTN estão aleatoriamente espalhados ao redor da área de simulação, com exceção do TLM, o qual permaneceu estacionário no centro da área. É importante ressaltar que nós pertencentes ao mesmo grupo não estão necessariamente próximos um do outro fisicamente. Portanto, o principal problema esperado nesse cenário é a dificuldade de um MLM (o gerente local do grupo) consultar os agentes que estão sob sua supervisão, uma vez que esses agentes estarão espalhados por toda a área de simulação, misturados com agentes pertencentes a outros grupos na rede. Como consequência, os MLMs terão problemas em reunir as respostas de todos os agentes do seu grupo para responder as requisições do TLM.

A Figura 2 (parte superior esquerda) apresenta a porcentagem de requisições do TLM que são efetivamente respondidas por um MLM (Taxa de Sucesso de *Polling*). No cenário onde dois grupos foram envolvidos, apenas 14,67% das requisições do TLM foram respondidas por MLMs. Na configuração com quatro grupos foi obtida uma taxa de resposta de 33,52%, e na configuração com seis grupos foi observado que 59,05% das requisições do TLM foram respondidas. Assim pode-se concluir que a efetividade da abordagem hierárquica aumenta à medida que a rede vai se tornando mais densa. Contudo, apesar de ser uma abordagem mais "distribuída" do que a centralizada, a taxa de requisições que foram respondidas de fato é inferior a taxa que foi observada na abordagem centralizada empregando o mesmo modelo de mobilidade.

O atraso médio registrado entre o TLM enviar uma requisição para um MLM e o TLM receber a resposta do MLM é registrada na Figura 2 (parte superior direita). No cenário com dois grupos, esse atraso médio foi de 9935 segundos. Quando quatro grupos foram simulados usando essa abordagem, o atraso médio registrado foi de 8538 segundos. Finalmente, foi registrado um atraso médio de 7175 segundos com seis grupos atuando no cenário.

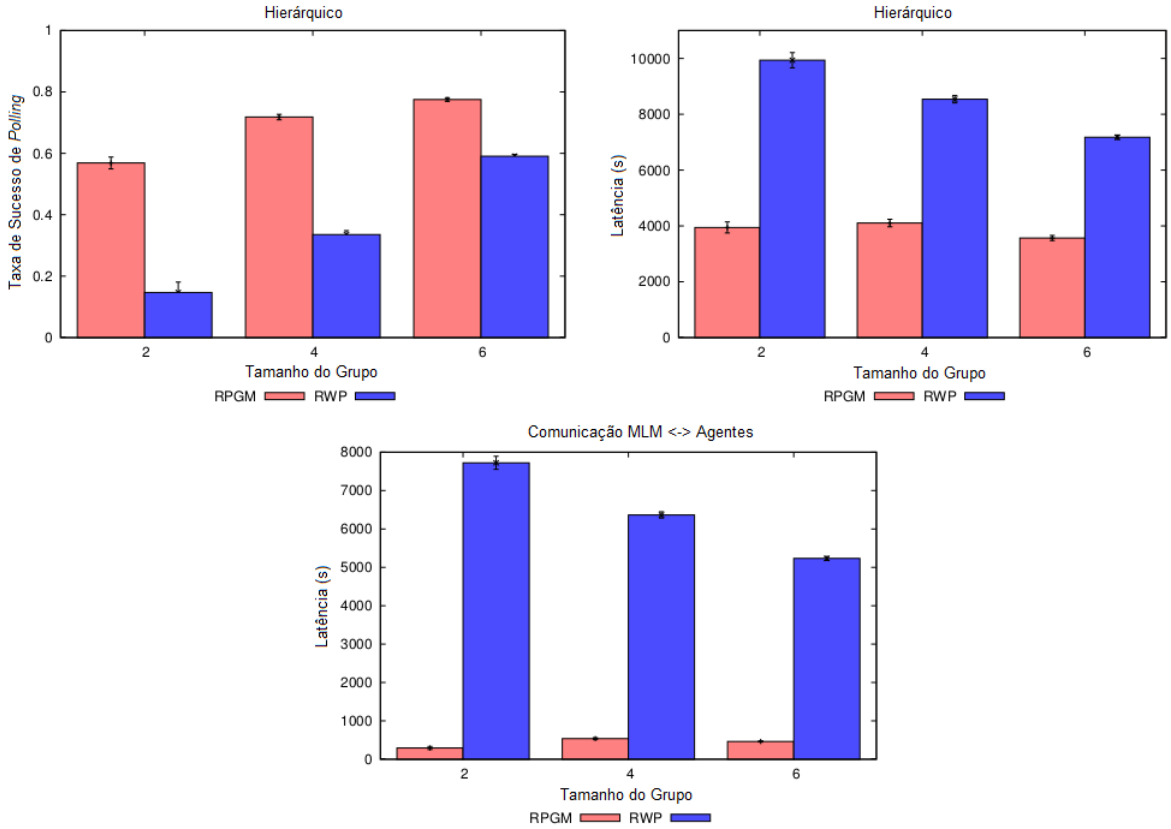


Figura 2. Análise das métricas na abordagem hierárquica

A Figura 2 (parte inferior) apresenta o atraso médio entre o momento em que o MLM recebe uma requisição do TLM e o momento em que ele reúne toda a informação necessário para responder a requisição do TLM. Ao medir esse tempo, pode-se estimar quando do atraso total envolvido no tempo de requisição-resposta do TLM é causado pelo MLM tentando reunir informação de todos os agentes do seu grupo. Nessa figura observa-se que MLMs na configuração com dois grupos apresentaram um atraso médio de 7723 segundos para responder a requisição do TLM. Na configuração com quatro grupos esse atraso médio foi de 6366 segundos, e na configuração com seis grupos esse atraso foi de 5232 segundos. Isso indica que, em todos os casos, ao menos 70% do atraso total envolvendo uma operação de requisição-resposta do TLM é causado pela dificuldade dos MLMs para consultar todos os agentes de seu grupo.

`Modelo RPGM`: Quando o modelo RPGM é aplicado nas simulações realizadas, cada conjunto de nós na rede passa a estar lógica e fisicamente agrupado. Essa forma de distribuição dos nós permite que os MLMs reúnam mais facilmente informação sobre os agentes que eles gerenciam por conta das características inerentes ao modelo de mobilidade empregado. Por outro lado, haverá uma maior quantidade de regiões vazias na área de simulação, o que tornará o roteamento inter-grupo mais desafiador.

A Figura 2 (parte superior esquerda) mostra que a quantidade de mensagens enviadas pelo TLM que foram efetivamente respondidas por MLMs (Taxa de Sucesso de *Polling*) aumentou em comparação aos cenários do modelo RWP. Na configuração com dois grupos foi observada uma taxa de sucesso de 56,86%. Já na configuração com quatro

grupos de nós, essa taxa de sucesso foi de 71,81%. Finalmente, observou-se uma taxa de sucesso de 77,52% na configuração com seis grupos. Também se percebeu que o número de requisições do TLM que foram respondidas aumentou à medida que a rede se tornou mais densa.

A latência média é significativamente reduzida quando o RPGM é usado, como se pode observar na Figura 2 (parte superior direita). O atraso médio quando dois grupos compõem o cenário de simulação é de 3943 segundos. Na configuração com quatro grupos o atraso médio é 4102 segundos, e quando o cenário possui seis grupos de nós o atraso médio é de 3565 segundos. Aqui não se observa muita variação à medida em que a rede se torna mais densa. Contudo, se observa a grande melhora quando se compara esses atrasos médios com aqueles registrados nas simulações empregando o modelo de mobilidade RWP. A análise da Figura 2 (parte inferior) revela que a causa disso é a dramática redução na quantidade de tempo que os MLMs gastam reunindo informação dos agentes sob suas responsabilidades. O tempo médio necessário para um MLM reunir informação de todos os agentes em seu grupo é de 293 segundos na configuração com dois grupos, 536 segundos na configuração com quatro grupos e 462 segundos na configuração com seis grupos. Esse tempo médio necessário para um MLM responder às requisições do TLM é menos de 15% do tempo médio do processo total de requisição-resposta do TLM.

4.3. Abordagem Distribuída

Uma abordagem de gerenciamento totalmente distribuída implica que todos os nós podem atuar tanto como gerentes quanto como agentes. Contudo, não existe muita informação acerca de implementações reais desse tipo de abordagem para o gerenciamento de redes tradicionais ou das chamadas "*challenged networks*". Por isso, foi proposta uma aplicação de gerenciamento distribuída simples, que não se baseia em nenhuma implementação real.

Na aplicação da abordagem distribuída empregada neste artigo, a cada vez em que dois nós fazem contato, eles fazem uma troca de mensagens requisitando o valor de um objeto específico da MIB do outro nó. No exemplo da rede de emergência utilizada neste artigo, os nós trocam dados sobre a quantidade de energia remanescente. À medida que um nó acumula contatos com outros nós ao longo do tempo, ele ajusta a sua potência de transmissão de forma a otimizar o uso de sua bateria sem penalizar demais a capacidade de comunicação da rede. Dessa forma, se um nó pertence a uma região da rede onde os vizinhos são numerosos e possuem bastante energia remanescente, esse nó pode minimizar sua potência de transmissão sem criar um grande "buraco" na DTN. Por outro lado, se esse nó não conseguir encontrar vizinhos suficientes e/ou esses vizinhos possuírem pouca energia remanescente na bateria, esse nó pode maximizar sua potência de transmissão de modo a aumentar a capacidade geral de comunicação da rede. Ao fazerem isso, cada nó está agindo como um gerente e como um agente, em um tipo de aplicação de gerenciamento sensível ao contexto.

A comunicação nessa aplicação apresenta a vantagem de confiar quase que completamente em comunicação local, sem depender de protocolos de roteamento DTN para a entrega de mensagens destinadas a porções da rede desconectadas ou de grandes atrasos. Contudo, a quantidade de informação que o nó possui para tomar decisões também é limitada ao número de encontros pelas quais esse nó passa na DTN. Assim, nessa abordagem distribuída é crucial que a aplicação de gerenciamento seja desenvolvida de modo

a considerar como um nó pode estimar o desempenho da rede DTN inteira conhecendo apenas informação relativa a parte da rede.

A seguir será avaliada a aplicação de gerenciamento distribuído sob os modelos de mobilidade RWP e RPGM.

`Modelo RWP`: Após o início da simulação, mensagens *get-request* são criadas apenas quando dois ou mais nós entram no raio de transmissão um do outro. Na maioria dos casos, a sessão de comunicação inteira entre dois nós ocorre antes desses nós saírem do alcance de comunicação do outro nó. Por conta desse comportamento, a taxa de sucesso do processo de *polling*, assim como os atrasos médios envolvidos na comunicação, é significativamente mais baixa do que nas outras duas abordagens de distribuição estudadas neste trabalho

Quase a totalidade das requisições foi corretamente respondida quando foi utilizada a abordagem distribuída nas simulações. Em outras palavras, a Taxa de Sucesso da *Polling* é próxima de 100% em todos os cenários simulados. Quando dois grupos de nós compuseram a DTN, 99,98% das requisições foram respondidas com sucesso. Nos casos de quatro e seis grupos foi obtida a mesma taxa de sucesso: 99,97%. As principais razões para explicar as mensagens não entregues nessa simulação é a ocorrência de contatos que ocorrem de forma extremamente efêmera (i.e., o nó mal cruza a fronteira do raio de transmissão do outro nó e em poucos instantes ele já se encontra fora do alcance de transmissão), como também os casos em que vários nós fazem contato um com o outro ao mesmo tempo, o que resulta na formação de filas que eventualmente podem não ser esvaziadas durante a duração do contato.

A Figura 3 apresenta o atraso médio registrado entre a emissão de uma mensagem *get-request* e a recepção da correspondente mensagem *get-response*. O tempo médio de resposta do processo de *polling* no cenário com dois grupos é 0,72 segundos, 1,21 segundos para o cenário com quatro grupos e 1 segundo para o cenário com seis grupos. Quando esses resultados são comparados com os resultados relacionados a outras abordagens, percebe-se uma dramática redução do atraso quando a abordagem distribuída é utilizada.

`Modelo RPGM`: No modelo RPGM, os nós DTN estão fisicamente agrupados, formando "ilhas" de nós na área de simulação. Essas concentrações tornam a ocorrência de contatos muito mais intensa em comparação com o uso do modelo de mobilidade RWP. Assim, é comum que mais de quatro ou cinco nós estejam em contato entre si ao mesmo tempo, o que leva a formação de filas nos dispositivos como o resultado da criação de múltiplas mensagens de resposta. Como algumas vezes a duração desses contatos não é suficiente para o esvaziamento dessas filas, algumas mensagens são entregues aos seus destinos em outras oportunidades, aumentando o tempo de resposta médio do processo de *polling*. Isso é observado na Figura 3, onde o atraso médio aumenta significativamente quando comparado aos cenários empregando o modelo RWP. Quando dois grupos de nós estão presentes na simulação, o atraso médio na comunicação requisição-resposta é 0,68s. Na configuração com quatro grupos, o atraso médio é de 9,37s, e na configuração com seis grupos é de 14,47s. Assim, uma alta concentração de nós pode levar a um aumento no tempo de resposta da aplicação de gerenciamento distribuído.

A taxa de sucesso de *polling* alcançada com o modelo RPGM praticamente não

apresentou diferença em comparação com os resultados obtidos com o modelo de mobilidade RWP. Na configuração com dois grupos, quase 100% das requisições foram corretamente respondidas. Na configuração com quatro grupos essa taxa foi de 99,89%, e na configuração com seis grupos ela foi de 99,73%.

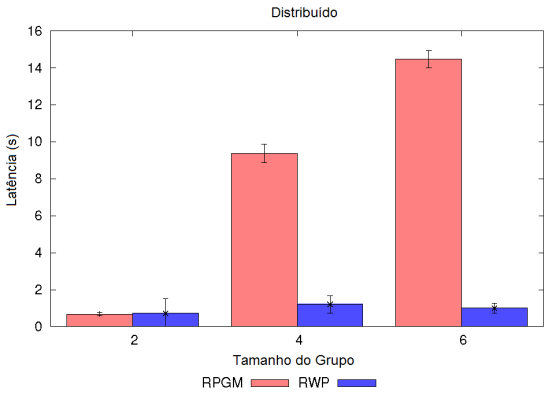


Figura 3. Atraso médio na abordagem distribuída.

5. Conclusões e Trabalhos Futuros

Esse artigo estudou o desempenho das abordagens de distribuição comumente encontradas em arquiteturas de gerenciamento de redes convencionais, de modo a identificar suas fraquezas e forças. Foram analisadas três abordagens: centralizada, hierárquica e distribuída. Até onde estão cientes os autores, esse é o primeiro estudo que avalia as abordagens de gerenciamento existentes em um ambiente DTN simulado. Esse é um passo muito importante em direção ao desenvolvimento de soluções de gerenciamento e tolerância a falhas para DTNs, uma vez que os resultados alcançados neste trabalho não apenas dão informação sobre quais abordagens de gerenciamento são viáveis para DTNs, como também destacam vários aspectos da operação de DTN que são necessários de serem levados em conta no desenvolvimento de uma solução de gerenciamento DTN.

Resultados das simulações mostraram que existe um balanço a ser considerado entre a quantidade de informação que um nó pode ter sobre as condições gerais da Rede Tolerante a Atrasos e Desconexões e a taxa de entrega de mensagens de gerenciamento. As abordagens centralizada e hierárquica devem ser usadas para adquirir dados de gerenciamento que não são sensíveis a longos atrasos. Se os dados de gerenciamento são sensíveis a atrasos, eles devem ser transmitidos e processados apenas em porções conectadas da DTN. Além disso, pode-se observar que a abordagem hierárquica é adequada apenas em DTNs onde os nós de um grupo estão concentrados próximos uns dos outros. Finalmente, os resultados não levam em consideração outras limitações como algoritmos de roteamento realistas, consumo de energia e buffers finitos. Deve-se avaliar o impacto dessas características em uma DTN para se estimar como os resultados apresentados neste trabalho serão influenciados.

Como trabalho futuro planeja-se estender esse estudo para outras tarefas de gerenciamento, tais como configuração de dispositivos e serviços, e alarmes. Também se planeja avaliar o impacto de outras características de DTNs em uma solução de gerenciamento, como algoritmos de roteamento, tamanho do buffer e overhead de mensagens de gerenciamento sobre a rede.

Referências

(2010). Wizzy digital courier. `http://www.wizzy.org.za/`.

Birrane, E. and Cole, R. G. Management of Disruption-Tolerant Networks: A Systems Engineering Approach. In *SpaceOps 2010 Conference*.

Calarco, G. and Casoni, M. (2011). Virtual networks and software router approach for wireless emergency networks design. In *73rd IEEE Vehicular Technology Conference*.

Case, J. D., Fedor, M. L., and Schoffstal, J. D. (1990). Simple network management procotol (SNMP). RFC 1157.

Cerf, V., Burleigh, S., Hooke, A., Torgerson, L., Durst, R., Scott, K., Fall, K., and Weiss, H. (2007). Delay-tolerant networking architecture. RFC 4838 (Informational).

Dong, F., Hu, Y., Tong, M., and Ran, X. (2009). Supporting emergency service by retasking delay-tolerant network architecture. In *5th International Conference on Mobile Ad-hoc and Sensor Networks*, pages 360 –365.

Fall, K. (2003). A delay-tolerant network architecture for challenged internets. In *ACM conference on Applications, technologies, architectures, and protocols for computer communications*, pages 27–34.

Hong, X., Gerla, M., Pei, G., and Chiang, C.-C. (1999). A group mobility model for ad hoc wireless networks. In *2nd ACM international workshop on Modeling, analysis and simulation of wireless and mobile systems*, pages 53–60.

Ivansic, W. (2009). DTN network management requirements. Internet Draft (Informational).

Jiang, P., Bigham, J., and Bodanese, E. (2011). Adaptive service provisioning for emergency communications with dtn. In *IEEE Wireless Communications and Networking Conference*, pages 2125 –2130.

Keränen, A., Ott, J., and Kärkkäinen, T. (2009). The ONE Simulator for DTN Protocol Evaluation. In *2nd International Conference on Simulation Tools and Techniques*.

Levi, D. and Schoenwaelder, J. (2001). Definitions of managed objects for the delagation of management scripts. RFC 3165.

Martin-Flatin, J.-P., Znaty, S., and Habaux, J.-P. (1999). A survey of distributed entreprise network and systems management paradigms. *Journal of Network and Systems Management*, 7(1):9–26.

Parikh, S. and Durst, R. C. (2005). Disruption tolerant networking for marine corps condor. In *IEEE Military Communications Conference*, volume 1, pages 325–330.

Rojas, A., Branch, P., and Armitage, G. (2005). Experimental validation of the random waypoint mobility model through a real world mobility trace for large geographical areas. In *8th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems*, pages 174–177.

Schönwalder, J., Quittek, J., and Kappler, C. (2000). Building distributed management applications with the ietf scriptmib. *IEEE Journal on Selected Areas in Communications*, 18(5):702–714.

Aplicação da IEC 61508 na Prototipação de Protocolos Seguros de Comunicação

William Vidal, Rodrigo Dobler, Sérgio Cechin, Taisy Weber e João Netto

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

{wrcvidal, rjdobler, cechin, taisy, netto}@inf.ufrgs.br

Resumo. *Visando garantir os níveis de segurança funcional adequados para a correta troca de informações entre os dispositivos de controle e instrumentação em uma planta industrial, as normas IEC 61508 e IEC 61784-3 impõem o uso de protocolos de comunicação seguros. A codificação desse tipo de protocolo deve ser realizada caso a caso e certificada para cada novo equipamento de automação. Neste contexto, este artigo apresenta um protótipo do protocolo PROFIsafe. A construção do protótipo partiu da especificação do protocolo e seguiu as recomendações da norma IEC 61508 para o desenvolvimento de software embarcado de segurança. O conjunto de regras e padrões de codificação desenvolvidos e a experiência na validação do protótipo serão aplicados no desenvolvimento de equipamentos para sistemas instrumentados de segurança.*

Abstract*. To ensure adequate integrity safety levels for the proper exchange of information between control and instrumentation devices in an industrial plant, IEC 61784-3 and IEC61508 standards impose the use of safe communication protocols. For each new equipment such protocols must be specially implemented and certified. In this context, this paper presents a prototype of PROFIsafe protocol. The prototyping started from the protocol specification and followed the recommendations of IEC 61508 for the development of safe embedded software. The set of coding rules we developed and our experience in the validation of the prototype will be incorporated in the development of equipment for safety instrumented systems.*

1. Introdução

Vivemos uma era de grandes desenvolvimentos tecnológicos, na qual se buscam novas tecnologias que possibilitem maior desempenho e eficiência nos processos industriais. Entretanto esse desenvolvimento não pode ser realizado sem a preocupação com a redução de riscos. Defeitos podem conduzir a acidentes graves que provocam perdas financeiras, afetem negativamente o meio ambiente ou causam danos à vida humana.

Sistemas computacionais são amplamente empregados para controle de processos críticos, para coleta e processamento de dados, para simulações e previsões, para gerência de instalações e para uma série de outras funções em ambientes industriais e empresariais. Sistemas computacionais são usados também para aumentar a segurança de instalações industriais executando as funções necessárias para prevenir ou mitigar acidentes nessas instalações (Leveson 1995). A correta operação do sistema em resposta a suas entradas, quando executando funções de segurança, está relacionada à segurança

funcional (Dunn 2003). Exemplos de funções de segurança são: controle de parada emergencial em indústrias químicas; bloqueio preventivo em maquinários pesados; sinais luminosos de alerta; alarmes de incêndio e detecção de gases tóxicos.

Um atributo importante para avaliar a segurança funcional (*functional safety*) é a sua integridade de segurança (Bell 2011) - *safety integrity*. Ela é definida como a probabilidade de um sistema realizar com sucesso a função de segurança a ele atribuída. No lugar de probabilidade de sucesso, o usual é mencionar a probabilidade de ocorrência de defeitos ao realizar a função de segurança. Dependendo do tipo de demanda da função de segurança, são usadas duas medidas para determinação do nível de integridade de segurança: probabilidade de defeito perigoso ao demandar a função (PFD), ou probabilidade de defeito perigoso por hora (PFH).

Para permitir a descentralização de funções nas plantas industriais (Thomesse 2005), sistemas de automação industrial, incluindo tanto controle como instrumentação de segurança, são baseados em barramentos de comunicação (Neumann 2007). A interação entre os elementos de uma rede de automação deve ser realizada de forma rápida, confiável e robusta. Protocolos de comunicação segura, tais como o FF SIS, openSafety, PROFIsafe e o Safety-over-EtherCAT, foram propostos para atingir esse objetivo.

Especificações dos protocolos estão disponíveis para que fornecedores possam desenvolver estes protocolos para as plataformas de hardware e software de seus produtos (Malik and Mühlfeld 2003). Não existem, entretanto, códigos prontos e pré-certificados. Para cada novo equipamento de um sistema de automação ou instrumentação, o protocolo escolhido deve ser codificado seguindo a especificação e obedecendo estritamente as recomendações da norma IEC 61508 (Bell 2006). Não basta provar a conformidade com a especificação. Todas as regras aplicáveis para um determinado nível de integridade de segurança devem ter sido seguidas para pleitear a certificação (Bilich and Hu 2009).

O objetivo do trabalho foi desenvolver o protótipo de um protocolo de segurança seguindo as normas cabíveis e estabelecer um conjunto de regras e restrições de codificação que auxilie na codificação de protocolos de segurança para diferentes tipos de plataformas de hardware e software. Como as regras são extremamente restritivas, elas formam uma lista de proibições e alternativas. O protótipo desenvolvido servirá posteriormente para aplicar as estratégias de validação recomendadas pela IEC 61508 e para treinar a equipe de desenvolvedores na empresa parceira quanto à aplicabilidade e cobertura de tais estratégias.

Entre os protocolos de comunicação segura, este trabalho focou o PROFIsafe, pois esse era o interesse da empresa parceira. O protocolo pode ser implementado a partir da sua especificação ou usando o pacote comercial PROFIsafe StarterKit. Em ambos os casos, o código deve ser adaptado para a plataforma final. Neste trabalho a implementação será efetuada a partir da especificação do protocolo.

O artigo apresenta um breve resumo da norma IEC 61508 e o protocolo PROFIsafe, as principais decisões de projeto sobre protótipo desenvolvido e os resultados alcançados.

2. Normas de segurança funcional

Nas últimas décadas, uma grande quantidade de normas de segurança foi proposta por organizações internacionais para vários setores de atividades (Esposito, Cotroneo, and Silva 2011). Algumas normas apresentam uma natureza mais genérica, como a IEC 61508 (Fowler and Bennett 2000). Outras são direcionadas para áreas específicas, como aviação, energia nuclear, máquinas, processos, indústria automobilística. Várias dessas normas específicas são derivadas da IEC 61508, incluindo a 61784-3, que padroniza perfis de comunicação segura.

2.1. IEC 61508

A IEC 61508 (IEC 1999) trata da segurança funcional de sistemas elétricos, eletrônicos e eletrônicos programáveis e serve como base para outras normas relacionadas com segurança (Bell 2006). O principal objetivo da norma é orientar o trabalho de equipes técnicas no desenvolvimento de equipamentos computacionais de segurança, visando alcançar níveis compatíveis com os exigidos por agências reguladoras em vários domínios de aplicação, incluindo os setores de óleo e gás, geração e distribuição de energia, máquinas e equipamentos, e transportes, entre outros (Gall and Wen 2010).

A norma cobre todo o projeto e desenvolvimento de hardware e software necessários para executar a função de segurança. São considerados o sensor, o controle lógico e o atuador final, incluindo os sistemas de comunicação. Adicionalmente, são consideradas quaisquer ações que possam ser realizadas por operadores humanos (Schonbeck, Rausand, and Rouvroye 2010).

Para classificar os sistemas, a norma especifica quatro níveis de desempenho para uma função de segurança, chamados de níveis de integridade de segurança, SIL (*Safety Integrity Level*). O nível mais baixo corresponde a SIL 1, enquanto o mais alto corresponde ao SIL 4. Os requisitos para um determinado SIL são mais rigorosos na medida em que se aumenta o SIL. Quanto maior o SIL, maior deve ser a garantia de uma menor probabilidade de ocorrência de defeitos perigosos.

Normas para segurança funcional têm em comum a obrigatoriedade ou a recomendação do uso de técnicas de prevenção e de tolerância a falhas para a redução de risco, além do projeto criterioso e documentado, tanto do hardware como de software, 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. Muitos acidentes devidos a falhas de software têm sido reportados na literatura em várias áreas críticas (Grottke and Trivedi 2007), (Zhivich and Cunningham 2009), (Hardy 2013), (Alemzadeh et al. 2013) onde os prejuízos à qualidade de vida são diretos e muitas vezes irremediáveis. Aplicar criteriosamente normas de segurança não fornece, entretanto, garantia absoluta de evitar acidentes provocados por falhas de software e hardware, mas ajuda a reduzir consideravelmente os riscos.

2.2. IEC 61784-3 e o conceito de Black Channel

Quando a função de segurança é realizada por um sistema de automação distribuído, com o uso da comunicação de dados, deve-se estimar a taxa de erros residuais deste

processo de comunicação e considerá-la na determinação das taxas de defeito daquela função de segurança como um todo. Para isso, a norma especifica que devem ser considerados erros como repetições, perda de mensagens, inserção de mensagens, mensagens entregues fora de ordem, corrupção, atraso e mascaramento, que correspondem a erros de endereçamento.

Alternativamente, pode-se utilizar a abordagem do canal escuro de comunicação (*Black Channel*). Este canal implementa um protocolo de comunicação de mais baixo nível, que não precisa seguir as recomendações da norma, e que suporta um protocolo de mais alto nível, este sim responsável pela segurança da comunicação. A principal vantagem do uso do *Black Channel* é que partes do canal de comunicação não precisam ser projetadas e validadas de acordo com a IEC 61508. Entretanto, conforme a IEC 61784-3, que trata dos perfis de comunicação de segurança funcional, mecanismos necessários para assegurar baixa taxa de defeitos devem ser implementados nos subsistemas relacionados com a segurança e que tenham interface com este canal de comunicação.

O conceito de *Black Channel* foi incorporado na versão de 2010 da IEC 61508, mas já aparecia em outras normas específicas de comunicação e publicações anteriores (Neumann 2007). O conceito é extremamente útil para o desenvolvedor de sistemas de segurança. É possível, desta forma, usar uma pilha convencional de protocolos de comunicação e se concentrar apenas na codificação do topo da pilha. Protocolos de comunicação seguros como EtherCAT, PROFIsafe, INTERBUS, OpenSafety e FOUNDATION Fieldbus se apoiam no conceito de *Black Channel* (Neumann 2007). Uma dada codificação desses protocolos é passível de ser certificada segundo a IEC 61508 até o nível SIL 3.

Perfis de protocolos seguros foram normalizados pela IEC 61784-3. A IEC 61784-3 baseia-se nas definições de segurança da IEC 61508. Ela estabelece que o *Black Channel* não deve consumir mais do que 1% do PFD ou do PFH, correspondente ao SIL projetado para a função de segurança.

3. Protocolo PROFIsafe

PROFIsafe é um protocolo de comunicação criado pela Profibus e Profinet Internacional. Sua especificação segue a norma IEC 61508. O nível de segurança que pode ser alcançado através PROFIsafe foi avaliado pela Agência de Inspeção Técnica alemã TÜV SÜD, que certificou a especificação do protocolo para aplicações até SIL 3. Deve ser observado que só a especificação está certificada. Cada nova codificação para uma dada plataforma deve passar por todo o processo de certificação para esta plataforma específica. Também não basta apenas usar um protocolo seguro para contar com um sistema de segurança, todos os componentes de hardware e software do sistema devem ser também certificados.

Com o protocolo PROFIsafe, as redes PROFIBUS e PROFINET, da Profibus e Profinet International, passaram a possibilitar que os seus equipamentos e sistemas pudessem ser usados na implementação de funções de segurança.

Muitos fabricantes de componentes seguros como, por exemplo, Siemens, WAGO, BECKHOFF, participaram da criação de padrões, o que permitiu o desenvolvimento de um portfólio extenso e completo de produtos (Malik and Mühlfeld 2003). Estes produtos são usados para realizar funções de segurança em processos

industriais, no setor de transportes, em equipamentos de guindastes, teleféricos, na indústria automotiva, entre outros. Como exemplo dos equipamentos desenvolvidos com PROFIsafe, pode-se citar: portas remotas de entrada e saída, sensores ópticos, sistemas de controle de segurança, gateways seguros, sensores de segurança, dispositivos com funções de segurança integrada, válvulas e bloqueios de válvulas.

3.1. Comunicação no PROFIsafe

O protocolo utiliza uma abordagem de canal simples e deve ser capaz de permitir o processamento nos equipamentos de controle, assim como a comunicação, de informações seguras e não seguras. Para isso, é necessário garantir que esses dois tipos de processamento ocorram de forma logicamente isolada. Essa abordagem permite a operação de sistemas que apresentam um único barramento físico de comunicação, o que é vantajoso do ponto de vista econômico.

O protocolo deve ser implementado sobre um *Black Channel*, o que o 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 construída sobre o canal inseguro.

O *Black Channel* torna desnecessária a avaliação de segurança dos elementos que o compõem: *backplanes* individuais, caminhos de transmissão e redes PROFIBUS e PROFINET. Toda a segurança do caminho, desde a geração do sinal seguro em um dispositivo remoto de entradas até o seu destino onde será processado, é obtida através dos mecanismos de proteção do protocolo seguro codificado sobre o Black Channel.

Na nomenclatura do PROFIsafe, existem três elementos: O F-Module, o F-Host (ou mestre) e o F-Device (ou escravo). Um módulo é representado por: entradas ou saídas digitais e analógicas, módulos de força, módulos de partida de motores e conversores de frequência, entre outros. Um mestre é um controlador. Um exemplo de dispositivo é um *scanner* a laser. Os parâmetros individuais de segurança variam de acordo com as diferentes tecnologias de segurança dos dispositivos e devem ser codificados e, principalmente, protegidos de operações que levem a defeitos no sistema.

3.2. Modelo de falhas

Erros na comunicação são ocasionados por falhas de hardware, interferência eletro-magnética e outros tipos de interferência ambiental. O modelo de falhas abrange erros de transmissão, repetições, perda de mensagens, inserção, sequenciamento, corrupção, atraso e mascaramento.

Erros de transmissão ocorrem quando causas externas ocasionam dados corrompidos. Corrupção é semelhante a erros de transmissão, mas tem como causa elementos do próprio sistema. Repetição ocorre quando mensagens velhas são repetidas. Um dispositivo de barramento com defeito pode apagar ou inserir uma mensagem ou modificar a ordem relativa das mensagens. Atrasos geralmente ocorrem quando os enlaces de comunicação estão saturados e as mensagens gastam mais tempo para circular do que o previsto. Mascaramento mistura mensagens relevantes e sem relevância para a segurança.

3.3. Mecanismos de detecção de erros

Para detectar e corrigir erros de comunicação, PROFIsafe foi especificado com os seguintes mecanismos: número sequencial de mensagens, controle de temporização, identificador único e CRC. Todos os mecanismos são tradicionais e aplicados largamente em inúmeros protocolos de comunicação. A diferença é a combinação dos diversos mecanismos.

O número de sequência permite identificar se as mensagens estão na ordem correta. O número de sequência detecta repetição não intencional, perda de mensagens, inserção de mensagens, sequência incorreta de mensagens e erros de repetição de mensagens. No protocolo, o número de sequência possui apenas um bit e tem que aparecer alternado entre duas mensagens. *Watchdog* detecta atrasos no caminho de comunicação. No protocolo, o *watchdog* detecta também falhas de perda de mensagens, mascaramento e inserções em mensagens.

O identificador único de conexão detecta mensagens mal encaminhadas entre transmissor e receptor. Esse identificador é conhecido unicamente pelos dois dispositivos envolvidos e é estabelecido a cada conexão. O identificador permite detectar endereçamento incorreto entre segmentos da rede, além de inserções e mascaramento em mensagens.

Finalmente, o CRC detecta bits de dados corrompidos nas mensagens. Adicionalmente, esta técnica permite detectar se algum módulo seguro realizou trocas de informações entre mensagens de segurança, e auxilia na detecção de mascaramento de mensagens.

No PROFIsafe, o CRC é calculado em múltiplos níveis. Primeiramente, ao longo da parametrização inicial do protocolo, é calculado o CRC1, composto de 2 octetos. O CRC, que é então encapsulado no pacote, é chamado de CRC2. Ele é formado pelos campos do pacote denominados dados de saída, F-Parametros, Status/Control Byte e Número Consecutivo. Para trazer mais segurança à técnica, o CRC é recalculado quando o pacote chega ao seu destino e comparado byte a byte com o campo CRC2 presente no pacote. Isso é possível, pois todos os bytes necessários para recalcular o CRC se encontram no próprio pacote ou já foram estabelecidos na fase inicial de parametrização do PROFIsafe.

4. Construção do protótipo

A empresa parceira no projeto pretende desenvolver uma linha de produtos de segurança certificados em conformidade com a IEC 61508. Para atingir tal objetivo é preciso adquirir experiência com a aplicação da norma, que é extensa, abrangendo mais do que 800 páginas de recomendações, sugestões e restrições para todas as fases do ciclo de vida de um produto de hardware e software. A certificação leva em conta não apenas o produto, mas toda a cultura de desenvolvimento e produção da empresa, portanto, o período de adaptação é longo.

O trabalho de prototipação irá fornecer subsídios para a empresa definir métodos e técnicas apropriados à produção de sistemas de segurança visando certificação de conformidade com a IEC61508. Assim, definiu-se que o protótipo do subsistema de comunicação seria um meio adequado para a aplicação das técnicas recomendadas de desenvolvimento e validação e permitiria adquirir experiência suficiente para estabe-

lecer padrões de codificação a serem seguidos pela empresa.

4.1. Sistemas de segurança versus sistemas computacionais convencionais

Há algumas diferenças entre o processo de desenvolvimento de software e hardware em projetos tradicionais e em projetos com requisitos de segurança. Nesses últimos, a criatividade do desenvolvedor é restrita a aplicar técnicas que já se mostraram adequadas na prática (Wassyng et al. 2011). Novidades acadêmicas e inovações não são bem acolhidas pelas normas de segurança funcional.

O principal desafio é aplicar técnicas de tolerância a falhas e engenharia de software no desenvolvimento de dispositivos de segurança levando criteriosamente em conta requisitos de segurança funcional e seguindo as normas apropriadas para o domínio da aplicação e sem explosão dos custos de desenvolvimento.

4.2. Modelo de comunicação

O sistema executa uma aplicação com dados recebidos pelos sensores e devolve os resultados para os atuadores. As portas de entrada (F-Devices) que coletam dados dos sensores se comunicam com o controlador (F-Host) através de um *Black Channel*. O controlador se comunica com as portas de saída (F-Devices) que fornecem dados aos atuadores através de outro *Black Channel*. O protocolo de comunicação é o PROFIsafe.

O modelo de comunicação do PROFIsafe é do tipo um para um (mestre/escravo), tendo o mestre como o F-Host e o escravo como o F-Device. O F-Device só envia mensagens em resposta às mensagens de solicitação de pedidos. Se o F-Device for de entrada, a resposta contém os dados de entrada. Se o F-Device for de saída, o dispositivo recebe os dados e confirma ao host o recebimento da mensagem. Como o dispositivo jamais gera uma mensagem espontaneamente e apenas responde a solicitações, no protótipo do PROFIsafe foi usada uma arquitetura sem threads, onde a aplicação determina o passo de execução de todos os mecanismos implementados na parte segura da arquitetura.

A discussão sobre a implementação nos próximos itens foca a codificação do protocolo como executada nos F-Devices. O F-Host tem uma implementação semelhante, mas como ele atua como mestre, seu código é mais complexo.

4.3. Arquitetura do sistema

O objetivo do protocolo é garantir a comunicação segura. Dessa forma, a arquitetura proposta inicialmente busca assegurar a separação entre a parte segura e a parte não segura.

A codificação do protocolo segue o conceito de *Black Channel.* Os mecanismos padrões de comunicação são componentes da parte não segura do sistema. O código da aplicação e uma porção do código do protocolo de comunicação ficam na parte segura. A codificação deve obedecer à norma: entre outras limitações pode-se citar a impossibilidade de uso de ponteiros (o que levou ao uso das chamadas de sinalização e envio de dados byte-a-byte) e de threads na parte segura (essa limitação pode ser removida, caso exista um sistema de gerência de threads que seja certificado).

Com isso, foi proposta uma arquitetura com duas partes e três camadas (Figura 1): a parte segura, onde está a aplicação; o código do protocolo seguro; e a parte não segura,

representando o início do *Black Channel*, onde está a biblioteca do canal de comunicação. Além disso, cada camada comunica-se com as camadas superiores ou inferiores através de um mecanismo de buffers de mensagens.

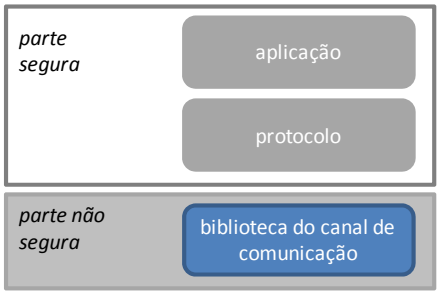


Figura 1 – Arquitetura de 3 camadas

4.4. Interação entre as partes seguras e não seguras

A troca de informação entre as partes segura e não segura não está claramente definida nas normas, e precisou ser desenvolvida. Nossa primeira abordagem está detalhada na Figura 2, onde os buffers de entrada e saída das mensagens estão nas camadas não segura e segura, respectivamente.

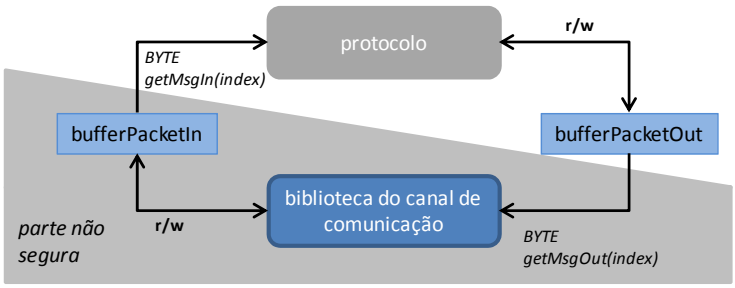


Figura 2 – arquitetura dos buffers de entrada e saída

Essa abordagem deixa o código com alto acoplamento. Além disso, ele fica pouco coeso e não padronizado. A escrita e leitura diretamente nos buffers requerem o posicionamento do código em uma das camadas (segura ou não segura) e o acesso aos buffers que estão na outra camada. Esta abordagem não oferece o isolamento de código requerido pelas normas, consequentemente, ela teve que ser aprimorada para garantir esse isolamento.

A solução foi restringir a parte não segura no que diz respeito ao acesso aos buffers internos da parte segura. Isso foi realizado por funções específicas de controle de acesso, usadas apenas para leitura.

Informações entre camadas de software do F-Device são trocadas através de chamadas de função. Conforme pode ser visto na Figura 3, são duas funções de "descida" e duas de "subida". Além disso, cada par possui uma função para a passagem de dados e outra para sinalização (que não têm parâmetros). O principal motivo para o desenvolvimento desta solução, além dos apurados anteriormente, foi a restrição do uso de ponteiros pela norma IEC 61508.

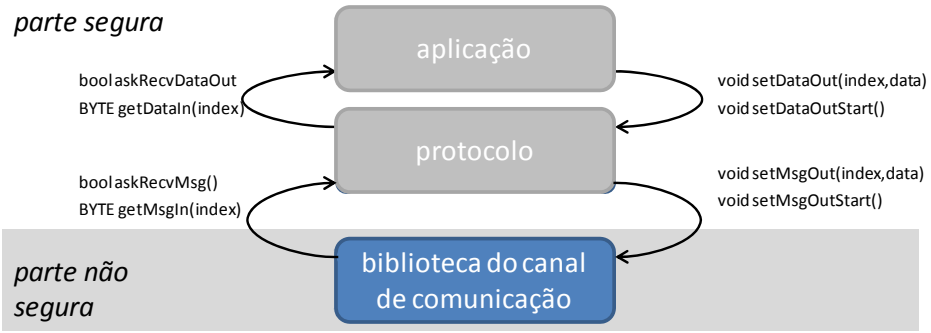


Figura 3 - arquitetura do protótipo

4.5. Recepção e envio de pacotes

O pacote PROFIsafe é formado por dados (até no máximo 123 bytes), CRC (3 ou 4 bytes) e *Status/Control Byte* (1 byte). Neste protótipo, optou-se por *payloads* de 12 bytes (correspondendo ao modo de operação rápido previsto na especificação), formando um pacote de 16 bytes.

Todo o envio de pacote pelo F-Device é precedido de uma recepção de pacote. Para envio de pacotes, a aplicação monta a sua mensagem através de setDataOut() que repassa os dados para a camada abaixo (camada de protocolo). Esses dados são salvos, byte-a-byte, em uma área de dados chamada de bufferDataAplicationSend. Em seguida, a aplicação sinaliza que terminou o envio dos dados através de setDataOutStart(). Com isso, o protocolo é notificado que os dados de entrada estão prontos para o seu uso exclusivo.

Prontamente, a camada de protocolo começa o encapsulamento do pacote. Os dados são transferidos para o módulo que calcula o CRC. Após, é criado o byte de controle/status que é encapsulado em bufferToSend para formar o pacote PROFIsafe. Após o encapsulamento, o pacote está pronto em bufferToSend que agora é transferido, byte-a-byte, para a camada inferior (denominada biblioteca) através da chamada setMsgOut(). A sinalização setMsgStart() avisa para a camada de biblioteca que os dados foram transferidos.

Nota-se que para a camada de protocolo, a sinalização que a mensagem foi enviada é a chamada *setMsgStart().* A camada de biblioteca funciona como uma caixa preta; não interessa de que forma a mensagem irá chegar ao seu destino.

4.6. Sistema de interrupção

O código do protocolo é embarcado em um módulo de hardware com poucos recursos computacionais. No protótipo foi necessário incorporar um sistema de interrupção, uma vez que não há um sistema operacional disponível no dispositivo que possa tratar as interrupções do hardware.

Não existe restrição ao uso de um sistema operacional embarcado. Entretanto essa opção implicaria em contar com um sistema operacional certificado, o que representaria um custo muito elevado para equipamentos de segurança simples como módulos de entrada e saída. Por essa razão, faz parte da arquitetura do protótipo um sistema de interrupção. Sua primeira função é a detecção de *timeout*, onde um contador

é decrementado (*watchdog*) e a sinalização ocorre quando o contador zera. A segunda função está relacionada à recepção do pacote. Foi preciso criar um semáforo binário para garantir a exclusividade de acesso ao sinal que informa o recebimento de uma nova mensagem. Este sinal é atualizado pelo sistema de interrupções. O sistema de interrupção opera como uma *thread*, responsável por realizar tais verificações. Esse mecanismo encontra-se na parte não segura da arquitetura, ou seja, está no *Black Channel*.

4.7. Testes com o protótipo

O código do protótipo foi validado por testes sob falhas. Foi usada como carga de trabalho a troca de mensagens entre os dispositivos mestre e escravo, onde foram injetados e detectados atrasos e corrupção de mensagens.

Nos testes realizados, em condições normais de operação, a troca de mensagens entre os dispositivos mestre e escravo ocorreu de maneira correta. Nos testes com injeção de atraso, quando foi inserido um atraso maior do que o valor máximo tolerado pelo protocolo para o recebimento de mensagens, este foi detectado pelo mecanismo de timeout. A corrupção de mensagens foi provocada pela alteração de um byte no pacote, a qual foi detectada no F-Device por meio de CRC.

Os testes realizados são ainda incompletos, não abrangendo todos os tipos de falhas necessários para ativar todos os mecanismos de detecção de erros do protocolo implementado, que estão previstas no modelo de falhas. Os próximos testes de injeção de falhas irão considerar o modelo de falhas completo.

5. Resultados

O processo de desenvolvimento teve que considerar os aspectos relativos à especificação do PROFIsafe e às recomendações da IEC 61508.

Notamos que a especificação do PROFIsafe não ajuda na forma como este deve ser integrado ao restante do sistema. Dessa forma, não basta apenas conceber a implementação do protocolo. É necessário considerar o restante do sistema, para obter o isolamento de software exigido pela IEC 61508. Na especificação do protocolo existe pouca informação sobre como ele deve ser acionado. Essa omissão é suprida nos kits de desenvolvimento, onde é definida uma API de acionamento do protocolo.

As recomendações impostas pela IEC 61508 implicam em uma mudança de cultura para desenvolvedores acostumados com sistemas de mais alto nível sem restrições de segurança. Seguir as normas é uma tarefa difícil e não intuitiva. Especificação, planejamento de testes (validação), desenvolvimento, integração, operação, manutenção (ou seja, procedimentos para modificação) e testes devem fazer parte do ciclo de vida de qualquer bom projeto de software, mesmo aqueles sem qualquer requisito de segurança. A diferença entre sistemas a serem certificados pela IEC 61508 e os demais é que a aplicação das técnicas e medidas recomendadas para cada fase do ciclo de vida deve ser justificada e demonstrada para uma equipe externa de certificadores através de documentação adequada.

A norma preconiza o estabelecimento de um padrão de codificação que deve ser obedecido pelos desenvolvedores. Segundo a IEC 61508, o padrão de codificação deve:

- Definir boas práticas de programação,

- Proibir características não seguras da linguagem,
- Promover facilidades para entender o código,
- Facilitar verificação e teste,
- Especificar procedimentos para documentação do código fonte,
- Garantir que no mínimo o código fonte deva conter: entidade legal, descrição, entradas e saídas, histórico da gerência de configuração.

O padrão de codificação proposto incorpora a experiência com o desenvolvimento do protótipo e consiste de uma lista de restrições e alternativas. Uma parte da lista é mostrada na Tabela 1.

Tabela 1 – Restrições e alternativas para a codificação de sistemas seguros

Não usar recursão
Não usar ponteiros
Não usar variáveis e objetos dinâmicos
Não usar conversão automática de tipos
Não usar desvios incondicionais em linguagens de alto nível
Usar abordagem modular
Usar bibliotecas de módulos e componentes de software confiáveis / verificados quando disponíveis
Limitar o número de parâmetros para as rotinas
Usar apenas um ponto de entrada e saída por rotina
Usar interfaces completamente definidas
Não usar linguagens orientadas a objeto
Evitar linguagens fracamente tipadas
Não usar estratégias de detecção de erros baseadas em inteligência artificial
Facilitar rastreabilidade
Favorecer testabilidade

As regras podem parecer muito restritivas e comprometer a produtividade de geração de código. Novas técnicas, métodos e estratégias para o desenvolvimento de software visando aumentar a produtividade aparecem continuamente e são rapidamente adotados pelas empresas desenvolvedoras. Métodos e técnicas estão sujeitos a um processo seletivo bem acentuado. Os melhores sobrevivem após algum tempo.

Muitos métodos de desenvolvimento promissores não resistem à passagem do tempo, e são substituídos por outros mais eficientes. Algumas técnicas inovadoras aplicadas ao software mostram posteriormente fragilidades em relação à robustez do código gerado. Assim o tempo é um importante aliado na escolha dos métodos e técnicas mais eficientes para a área de desenvolvimento de software relacionado à segurança. Métodos e técnicas que não podem ser provados seguros ou que geram software com maior probabilidade de conter falhas residuais não devem ser aplicados no desenvolvimento de software que executa funções de segurança. Métodos e técnicas mais antigos, que na prática já mostraram capacidade de gerar produtos robustos, são preferidos nesta área. Esse é o motivo porque as regras de codificação que aparecem na Tabela 1 parecem tão restritivas. Elas sozinhas não garantem a segurança do código, mas permitem gerar código com menor probabilidade de erros.

A maior parte das restrições não é novidade para os desenvolvedores. Apesar disso, convencer desenvolvedores a não usar ponteiros e evitar linguagens orientadas a objetos e até linguagens fracamente tipadas é uma tarefa difícil. Mas evidências baseadas na experiência prometem ser uma boa motivação.

6. Conclusão

O protocolo de comunicação segura escolhido atende as recomendações de segurança da norma IEC 61508, tendo sua especificação sido certificada para aplicações de segurança até SIL 3. Utiliza o conceito de *Black Channel*, no qual os recursos necessários para transportar as mensagens do protocolo seguro não são considerados para fins de avaliação da segurança. Além disso, o protocolo apresenta mecanismos de controle de falhas adequados aos modelos de falhas descritos na IEC 61508. Esses mecanismos são o número de sequência, *watchdog*, identificadores únicos e CRC.

A experiência no desenvolvimento do protótipo gerou um padrão de codificação exigido pela norma. O protótipo permitirá também aplicar as estratégias de teste previstas na norma. Testes iniciais sob falhas já foram conduzidos. Uma experiência mais abrangente com testes do protótipo será repassada à empresa interessada para ser aplicado no desenvolvimento de seus produtos. Caso a empresa opte por desenvolver sua própria codificação do protocolo de comunicação, o protótipo será usado para comparação dos resultados entre as duas implementações.

Agradecimentos

O projeto foi apoiado financeiramente pelo programa de formação de recursos humanos da ANP para o setor de Petróleo e Gás - PRH PB-217 (Petrobras) e o projeto Rio-SIL (Finep).

Referências

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.

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.

Grottke, Michael, and Kishor S. Trivedi. 2007. "Fighting Bugs: Remove, Retry, Replicate, and Rejuvenate." *Computer* 40 (2): 107–9.

Hardy, Terry. 2013. "Case Studies in Process Safety: Lessons Learned From Software-Related Accidents."

IEC. 1999. *IEC 61508: Functional Safety of Electrical, Electronic/Programmable Electronic Safety-Related Systems*.

Leveson, Nancy G. 1995. *Safeware: System Safety and Computers*. 1 edition. Reading, Mass: Addison-Wesley Professional.

Malik, Robi, and Reinhard Mühlfeld. 2003. "A Case Study in Verification of UML Statecharts: The PROFIsafe Protocol." *J. UCS* 9 (2): 138–51.

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

Schonbeck, Martin, Marvin Rausand, and Jan Rouvroye. 2010. "Human and Organisational Factors in the Operational Phase of Safety Instrumented Systems: A New Approach." *Safety Science* 48 (3): 310–18.

Thomesse, J.-P. 2005. "Fieldbus Technology in Industrial Automation." *Proceedings of the IEEE* 93 (6): 1073–1101.

Wassyng, Alan, Tom Maibaum, Mark Lawford, and Hans Bherer. 2011. "Software Certification: Is There a Case Against Safety Cases?" In *Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems*, 206–27. Springer.

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

Uma Análise da Sobrecarga Imposta pelo Mecanismo de Replicação de Máquinas Virtuais Remus

Marcelo Pereira da Silva, Guilherme Koslovski*, Rafael R. Obelheiro*

Programa de Pós-Graduação em Computação Aplicada (PPGCA)
Universidade do Estado de Santa Catarina (UDESC) – Joinville, SC – Brasil

pereira@joinville.udesc.br, *{firstname.lastname}@udesc.br

Abstract. *Remus is a primary-backup replication mechanism for the Xen hypervisor, providing high availability to virtual machines by frequent checkpointing (on the order of tens of checkpoints per second). Despite providing good fault tolerance for crash and omission failures, the performance implications of using Remus are not well understood. We experimentally characterize Remus performance under different scenarios and show that (i) the checkpointing frequency that provides the best performance is highly dependent on application behavior, and (ii) the overhead might be prohibitive for latency-sensitive network applications.*

Resumo. *Remus é um mecanismo de replicação primário-backup para o hipervisor Xen, que fornece alta disponibilidade a máquinas virtuais realizando checkpoints com frequência (na ordem de dezenas de checkpoints por segundo). Apesar de oferecer boa tolerância a faltas para falhas de parada e omissão, as implicações de desempenho do Remus não são bem compreendidas. Este trabalho caracteriza experimentalmente o desempenho do Remus em diferentes cenários e mostra que (i) a frequência de checkpointing que resulta no melhor desempenho é altamente dependente do comportamento das aplicações e (ii) o overhead pode ser proibitivo para aplicações de rede sensíveis a latência.*

1. Introdução

Eventuais faltas em recursos computacionais, de comunicação e armazenamento podem comprometer a execução de uma aplicação ou até mesmo interrompê-la. Este fato motiva a constante busca por soluções computacionais (algoritmos, arquiteturas, protocolos e infraestruturas) com alta disponibilidade. Pesquisadores e projetistas de *software* têm criando soluções para o desenvolvimento de aplicações confiáveis, capazes de sobreviver a interrupções no substrato computacional. Infelizmente, esta tarefa é complexa e usualmente as soluções encontradas aumentam o tempo de desenvolvimento (inclusão de verificações de estado, *checkpoints*, entre outros), impondo ainda uma sobrecarga de processamento perceptível ao usuário da aplicação.

Paralelamente, os centros de processamento de dados têm explorado as vantagens introduzidas pela virtualização de recursos computacionais, tais como consolidação de servidores, economia no consumo de energia, e gerenciamento facilitado [Pearce et al., 2013]. Neste contexto, um usuário executa sua aplicação sobre

um conjunto de máquinas virtuais (MVs), ou seja, ocorre a introdução de uma abstração entre o *hardware* físico (que hospeda a MV) e a aplicação final. A camada de abstração, chamada de monitor de máquinas virtuais (MMV) ou hipervisor, tem acesso a todo o estado de execução de uma MV, e por isso pode ser explorada para introduzir tolerância a faltas de forma transparente para o *software* na máquina virtual [Bressoud e Schneider, 1996]. Essa proposta pioneira possuía diversas restrições de natureza prática, como fraco desempenho e disponibilidade limitada da tecnologia de virtualização na época. Mais recentemente, na esteira da ressurgência da virtualização, surgiu o Remus [Cully et al., 2008], um mecanismo para replicação de máquinas virtuais baseado no hipervisor Xen [Barham et al, 2003]. O Remus encapsula a aplicação do usuário em uma MV e realiza *checkpoints* assíncronos entre o hospedeiro principal e a cópia de segurança, com uma frequência na ordem de dezenas de *checkpoints (cps)* por segundo. Quando um problema ocorre no hospedeiro principal, a MV no hospedeiro secundário é automaticamente iniciada, o que induz um baixo tempo de indisponibilidade.

O grande atrativo do uso de replicação de MVs para prover tolerância a faltas é a transparência, uma vez que as aplicações e sistemas operacionais existentes podem executar sem modificações. Entretanto, é latente a necessidade de investigar qual a sobrecarga imposta por este mecanismo nas aplicações hospedadas. Cully et al. (2008) observaram que o tempo de indisponibilidade depende do comportamento da aplicação em execução, já que o estado da MV é constantemente migrado (migração ativa) entre os hospedeiros [Clark et al., 2005]. Complementarmente, o presente trabalho destaca duas variáveis visando contribuir com esta caracterização: i) o intervalo de *checkpoint* necessário para manter o nível de confiabilidade do serviço, e ii) o consumo de recursos computacionais e de comunicação. O impacto dessas variáveis pode ser percebido na visão do usuário (tempo de execução e latência na resposta a uma solicitação) e do administrador (custo de provisionamento e largura de banda necessária). *Através de uma análise experimental em um ambiente controlado, este trabalho caracteriza o comportamento destas variáveis para um conjunto de aplicações executadas em uma infraestrutura virtualizada na qual a alta disponibilidade é garantida pelo Remus*. Mais especificamente, contribuímos demonstrando experimentalmente que não existe um intervalo entre *checkpoints* eficiente para todas as aplicações, sendo este o principal responsável pela latência percebida pelo cliente. Em nosso entendimento, este trabalho apresenta a primeira análise experimental que quantifica estas variáveis.

O restante deste artigo está organizado da seguinte forma: a Seção 2 detalha o mecanismo de replicação de MVs implementado pelo Remus. A Seção 3 apresenta a metodologia experimental e os resultados obtidos. A Seção 4 discute os trabalhos relacionados. Por fim, a Seção 5 conclui o artigo.

2. O Mecanismo de Replicação Remus

Remus é um mecanismo para replicação de MVs que visa fornecer alta disponibilidade a um baixo custo. Basicamente, Remus efetua uma migração constante de uma MV entre hospedeiros físicos, de forma transparente ao usuário final. Quando uma falta ocorre no hospedeiro principal (faltas como de *hardware*, rede, queda de energia, entre outros), o mecanismo ativa a MV secundária sem perda de conexão na visão do serviço hospedado na MV. O Remus é totalmente agnóstico em relação ao sistema operacional em execução na MV, o qual é executado sem precisar de configuração adicional.

Nativo nas versões atuais do MMV Xen [Barham et al., 2003], Remus utiliza a técnica de replicar o estado/disco da MV entre hospedeiros físicos a uma alta frequência (dezenas de *cps/s*). Cada transferência contém a variação das informações da MV desde o último *checkpoint*, ou seja, mantém o estado de conexões e monitora as transações em memória e disco, bem como o estado atual da CPU virtual. O Remus segue o modelo de replicação passiva ou primário-*backup* [Budhiraja et al, 1993]: normalmente, os clientes interagem com a MV hospedada no primário, a qual é replicada no hospedeiro *backup*. Caso ocorra a falha do primário, a MV no *backup* passa a responder pelo serviço. No Remus, os papéis de primário e *backup* são estáticos: quando um primário que havia falhado é restaurado, ele é atualizado a partir do *backup* e reassume o papel de primário. Por definição, Remus oferece alta disponibilidade para MVs sujeitas a faltas de parada (*crash*) e omissão de recursos virtuais ou físicos; em caso de parada simultânea do primário e do *backup*, as MVs são deixadas em um estado consistente.

O mecanismo de replicação pode ser visto como uma versão modificada do procedimento para migração ativa (*live migration)* [Clark et al., 2005], nativamente oferecido pelo Xen. A migração permite mover uma MV de um hospedeiro para outro, sem reiniciá-la, com um pequeno tempo de inatividade. Este procedimento é comumente utilizado para facilitar o gerenciamento e o balanceamento de carga em ambientes virtualizados, mas não oferece alta disponibilidade. Há semelhanças nos processos de replicação e migração. O *live migration* realiza uma transferência inicial da memória da MV em execução e, posteriormente, realiza cópias iterativas, copiando somente o que foi alterado na memória desde a cópia anterior. Esse ciclo, que ocorre diversas vezes, é repetido até que exista uma quantidade pequena de memória a ser transferida. Neste momento, ocorre a etapa chamada de *stop-and-copy*: a MV, que ainda está em execução no hospedeiro original, é parada definitivamente, e todo o restante da memória ainda não migrada é então transferido para o hospedeiro de destino. Na próxima fase a MV é finalmente iniciada no novo hospedeiro e uma resposta ARP não solicitada [Stevens, 1994] é difundida na rede local, permitindo que ela assuma o endereço IP da MV no hospedeiro original. *Live migration* e Remus confiam em protocolos de camadas superiores para reenvio de tráfego de rede eventualmente perdido durante esse processo.

O processo de replicação do Remus, baseado em *live migration*, é composto por vários *cps/s*, sendo que cada *checkpoint* realiza um procedimento *stop-and-copy*. Mais especificamente, a cada *checkpoint* a MV sofre uma pausa e o conteúdo da memória é transferido não diretamente para o hospedeiro de *backup*, mas para um *buffer* local. Em seguida, a MV retoma sua execução, e o *buffer* é então transferido para o hospedeiro de *backup*. Neste momento, o estado de execução na MV é especulativo no hospedeiro primário: a MV permanece em execução, porém o tráfego de rede que sai da MV (presumivelmente contendo respostas às requisições dos clientes) fica retido em outro *buffer*, o de rede. Somente depois que o hospedeiro de *backup* confirmar o recebimento dos dados (ou seja, o *buffer* de memória foi totalmente recebido e o disco está sincronizado) é que o tráfego de saída é liberado para deixar o *buffer* de rede. A Figura 1 ilustra este cenário. As requisições (RX) enviadas pelo cliente sempre são recebidas pela MV, enquanto as respostas (TX) são enviadas somente após a confirmação de que as réplicas estão em um estado consistente, sincronizadas entre os dois hospedeiros. É importante ressaltar que os dados temporariamente armazenados nos *buffers* local e de

rede serão descartados pelo hospedeiro *backup* caso ocorra uma falha no primário antes do término de um *checkpoint*.

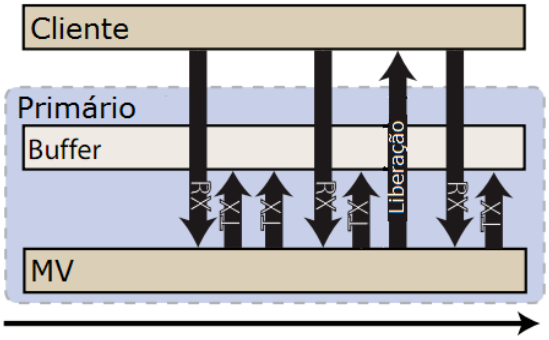


Figura 1. Funcionamento do *buffer* de rede. Adaptado de [Cully et al., 2008].

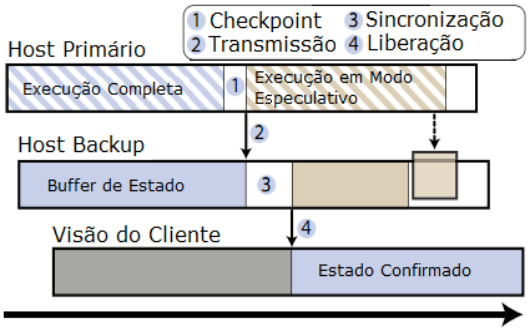


Figura 2. Modo especulativo e assíncrono. Adaptado de [Cully et al., 2008].

O procedimento de replicação alterna o estado de execução da MV entre especulativo e completo. A Figura 2 ilustra este processo: a interação com o cliente ocorre sempre de uma forma atrasada. Desta forma, o mecanismo garante que a memória e o disco da MV estão em estado consistente nos hospedeiros primário e de *backup*. A MV no hospedeiro *backup* mantém-se constantemente em estado de pausa (com execução suspensa), e somente executa se houver uma falha do primário. Isso difere da proposta de Bressoud e Schneider. (1996), onde o *backup* está sempre ativo, executando as mesmas instruções do primário (o que dificulta a garantia do determinismo das réplicas).

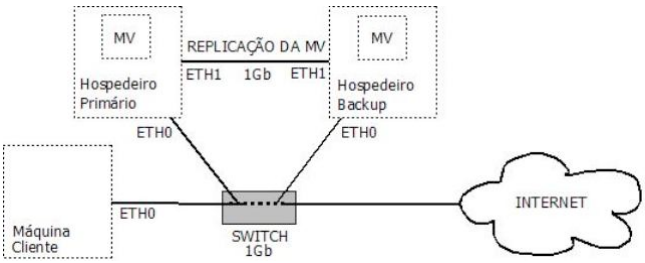


Figura 3. Remus em execução em uma rede local.

A replicação do disco é auxiliada pela ferramenta DRBD [Reisner e Ellenberg, 2005], configurada nos dois hospedeiros. O DRBD fornece um dispositivo virtual para replicação de blocos de disco, atuando entre a área de escalonamento de E/S e a camada de rede, sem conhecimento ou interação com o sistema de arquivos do disco. Na prática, ele oferece espelhamento de discos (RAID-1) via rede. Remus estendeu o protocolo utilizado pelo DRBD para suportar replicação com *cps* assíncronos. Sob Remus, DRBD opera em modo *dual-primary*, permitindo que o mecanismo de Remus decida qual hospedeiro é o primário.

A Figura 3 ilustra o funcionamento do Remus em uma rede local. Neste cenário, quando o primário falha, imediatamente o *backup* ativa a sua cópia da MV, atualizada até o último *checkpoint*. O *backup* detecta a falha do primário por *timeout* na recepção de *cps*; o primário, por sua vez, detecta a falha do backup por *timeout* no reconhecimento de *cps* (quando a falha do *backup* é detectada, o primário desativa a replicação). Assim, os *cps* e reconhecimentos funcionam como mensagens de *heartbeat*.

Após o problema com o hospedeiro primário ser resolvido, o cenário pode retornar à sua configuração inicial, migrando a MV do *backup* para o primário, reestabelecendo o mecanismo de proteção.

3. Análise experimental

Os *benchmarks* executados na análise experimental de Remus [Cully et al., 2008] quantificaram o impacto causado pelo mecanismo de replicação na execução de aplicações considerando a variação de latência no substrato físico. Neste cenário original, os autores não investigaram qual o intervalo de *checkpointing* apropriado para aplicações interativas e não interativas. Especificamente considerando aplicações interativas (sensíveis a latência), os autores sugerem a possível existência de sobrecarga, mas não quantificaram esta afirmação. Ainda, é importante ressaltar que algumas das otimizações listadas como trabalhos futuros no artigo original já foram implementadas (*e.g.*, compressão de páginas), o que sugere uma alteração (menor fluxo de dados entre primário e *backup*) no mecanismo de replicação. Mesmo que a compressão de dados resulte em um maior uso da CPU, ela ocorre no hospedeiro, não na MV protegida. Assim, analisamos o impacto no desempenho do uso do mecanismo de replicação Remus em sua versão atual, dividindo os testes em dois cenários: A) serviços executando isoladamente em uma MV; e B) e serviços que mantêm uma conexão permanente entre a MV e clientes externos.

Cenário A – Nesse caso, a sobrecarga imposta pela necessidade de parar e reiniciar a MV constantemente é relevante. Analisamos o tempo de execução de testes com operações específicas sobre a memória, dispositivos de E/S e CPU. Além da medição de desempenho, o fluxo de rede entre os hospedeiros físicos foi monitorado para identificar a utilização de largura de banda. Em suma, *o objetivo deste experimento é investigar o impacto causado pelo Remus considerando diferentes intervalos de checkpointing*. Inicialmente a ferramenta de *benchmark* Sysbench[1] foi utilizada para gerar cargas de teste para memória, disco e operações de entrada e saída. Em um segundo momento, um processo de compilação foi executado.

Cenário B – *Esse experimento investiga o impacto da replicação na latência observada pelos clientes de serviços em execução na MV*. Foram executados testes para descobrir qual o intervalo de *checkpoint/s* que causa um menor impacto à MV considerando aplicações com conexões TCP de longa duração.

3.1 Ambiente de testes

O ambiente de testes compreendeu três computadores com processador AMD Phenom II X4 (4 cores) de 2,8 GHz, memória RAM de 4 GB, disco 500GB SATA, executando o sistema operacional Ubuntu 12.10 (*kernel* 3.5.0-17-generic), e interconectados em uma rede local, conforme descrito pela Figura 3. O equipamento identificado como "Máquina Cliente" foi utilizado com origem das requisições para o cenário B. Os hospedeiros primário e de *backup* possuem duas interfaces de rede, sendo uma para a conexão com a máquina cliente através de um comutador de 1 Gbps, e outra exclusiva para a replicação da MV (1 Gbps via cabo *crossover*). Cada hospedeiro é virtualizado

[1] Disponível em http://sysbench.sourceforge.net.

com o MMV Xen 4.2.1 e protegido pelo mecanismo Remus nativo, executando DRBD versão 8.3.11-remus. A MV a ser replicada é composta por duas VCPUs, 20 GB de disco e 1 GB de memória. Seu sistema operacional é o OpenSUSE 12.2 (x86_64).

Para todos os testes realizados, a frequência de *checkpointing* foi definida em 10, 20, 30 ou 40 *cps/s*, seguindo estudos previamente realizados [Cully et al., 2008], [Rajagopalan et al., 2012]. Para cada experimento foram realizadas cinco rodadas, com coeficiente de variação máximo (entre todos os experimentos) de 6,4%.

3.2 Cenário A: CPU

Para investigar a sobrecarga do mecanismo de replicação em aplicações *CPU-bound*, utilizamos o *benchmark sysbench*, especificamente o módulo que calcula os números primos possíveis menores do que 10000. O tempo de execução pode ser comparado à execução sem Remus (identificado pela legenda 0).

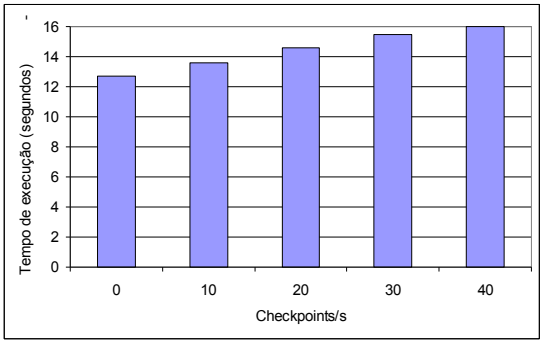


Figura 4. Tempo de execução do *benchmark* de CPU.

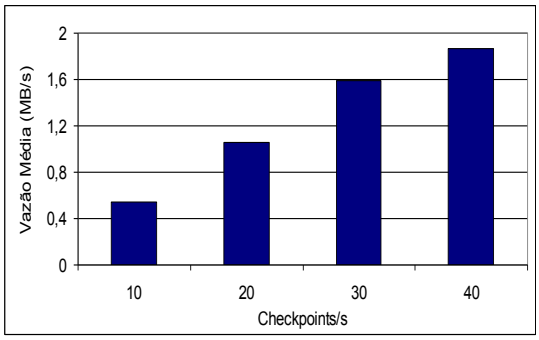


Figura 5. Vazão média de dados entre os hospedeiros durante a execução do *benchmark* de CPU.

Como mostra a Figura 4, o *overhead* causado pela replicação variou pouco em relação ao número de *cps*. Enquanto sem replicação o tempo foi de 12,7 s, com 10 *cps/s* foi de 13,6 s, resultando em 7% de sobrecarga no desempenho da MV. No pior caso, com 40 *cps/s*, a sobrecarga foi de 25,9%. Ou seja, conforme já observado por Cully et al. (2008), uma maior frequência de *cps* induz uma maior sobrecarga no desempenho de aplicações *CPU-bound*. Complementarmente, a Figura 5 ilustra a média da vazão de dados entre os hospedeiros primário e *backup* durante a replicação. Percebe-se que quanto maior o intervalo entre *cps*, melhor o desempenho (Fig. 4), e menor o fluxo de dados durante a replicação (Fig. 5). A baixa vazão é reflexo da otimização de compressão de dados de *checkpoint* implementada nas versões atuais do Remus.

3.3 Cenário A: memória

Este teste simula uma aplicação com uso intensivo de operações na memória principal. Para tal, configuramos o módulo de memória do *benchmark* sysbench para realizar escrita em memória até 10 GB, como blocos de 1 KB. Os resultados são comparados com uma execução sem Remus (barra rotulada com 0 *cps/s*).

Como mostra a Figura 6, a sobrecarga causada pelo Remus, em seu melhor resultado (com 10 *cps/s*), foi superior a 100% em relação à execução em uma MV desprotegida. É fato que esta sobrecarga pode ser um fator impeditivo para este tipo de aplicações. Novamente, observa-se que, quanto maior o intervalo entre *cps*, menor a sobrecarga, e

menor o fluxo de dados entre os hospedeiros (Figura 7). A sobrecarga é justificada pela natureza do mecanismo de replicação, que transfere, constantemente, todas páginas de memória alteradas desde o último *checkpoint* confirmado.

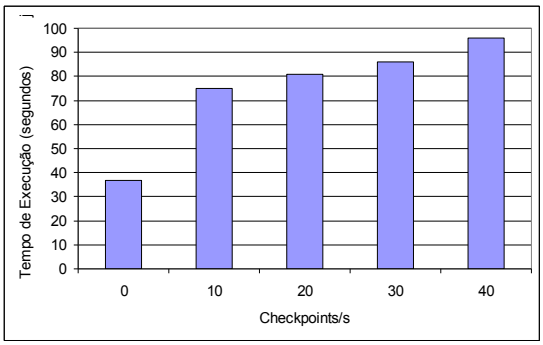


Figura 6. Tempo de execução do *benchmark* de memória.

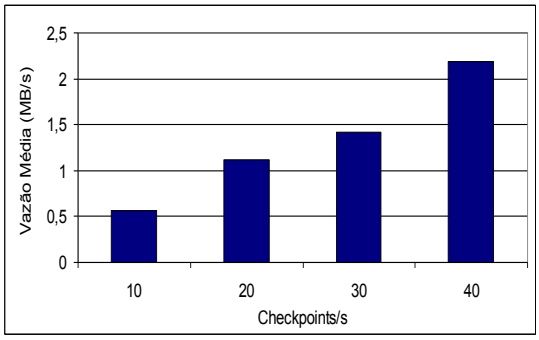


Figura 7. Vazão média de dados entre os hospedeiros durante a execução do *benchmark* de memória.

3.4 Cenário A: operações de entrada e saída

O teste executado realiza operações de leitura e escrita, aleatoriamente, no disco local da MV, sincronizado pelo mecanismo DRBD. Assim como nos testes anteriores, quanto menor a frequência de *checkpointing*, melhor o desempenho da MV (Figura 8) e menor o fluxo de dados transmitido entre os hospedeiros primário e *backup*, como mostra a Figura 9. O pior caso identificado foi com 30 *cps/s,* sendo 28,7% a sobrecarga observada (a pequena variação observada entre 30 e 40 *cps/s* segue o comportamento observado em alguns experimentos originais do Remus [Cully et al., 2008]). Se comparado com o valor base (sem Remus) e sobretudo com o *benchmark* de memória, onde o melhor intervalo de *cps/s* causou uma sobrecarga superior a 100%, observa-se que a sobrecarga de operações de E/S é tolerável para este tipo de aplicação. A vazão de dados entre os hospedeiros (Figura 9) segue o padrão dos demais experimentos.

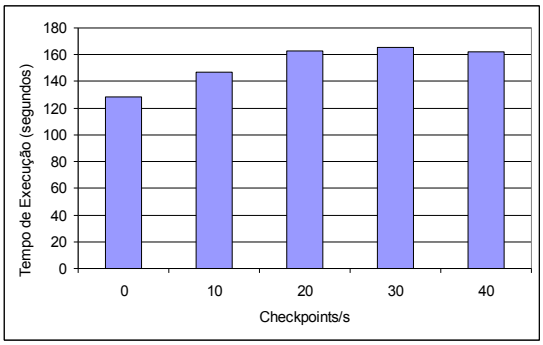


Figura 8. Tempo de execução do *benchmark* de operações de entrada e saída.

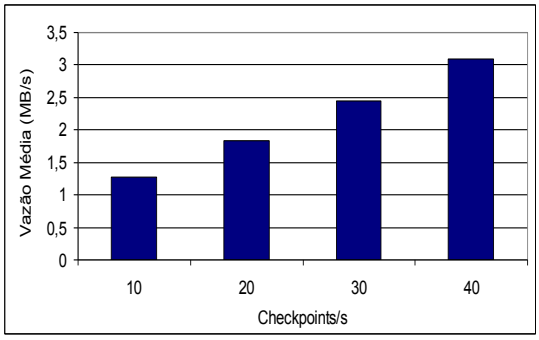


Figura 9. Vazão média de dados entre os hospedeiros durante a execução do *benchmark* de E/S.

3.5 Cenário A: compilação

Neste experimento efetuamos a compilação do BIND versão 9.9.4.p2[2] sobre uma MV replicada. Este processo envolve uso de CPU, memória, acesso a disco e rede (tráfego entre hospedeiros), ou seja, é uma combinação dos cenários individualmente analisados nas seções anteriores. Durante a compilação, 360 novos arquivos serão criados, totalizando 235MB, que deverão ser replicados para o *backup*. A Figura 10 apresenta o tempo de execução enquanto a Figura 11 mostra a vazão média de dados transferidos entre os hospedeiros.

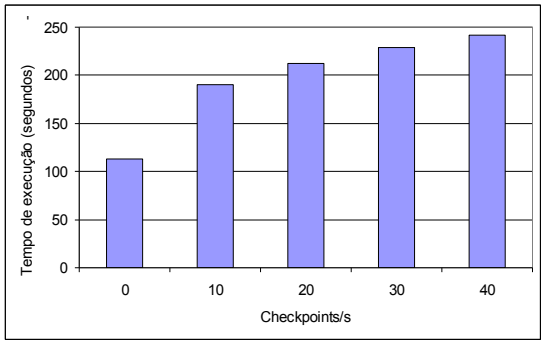


Figura 10. Tempo de execução da compilação do BIND.

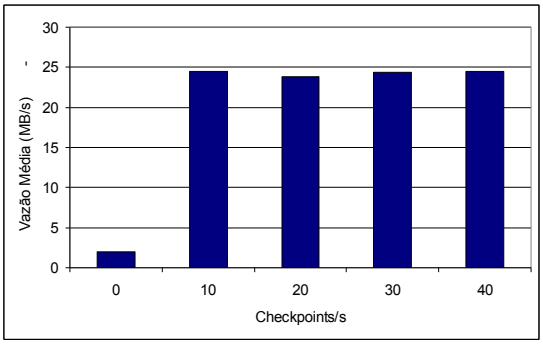


Figura 11. Média da vazão de dados entre os hospedeiros primário e *backup* durante a compilação

A sobrecarga imposta pela replicação foi de 68% para o melhor caso (10 *cps/s*). A sobrecarga identificada segue o comportamento dos testes anteriores, mas percebe-se que, embora a compilação seja uma aplicação com carga combinada de CPU, memória e entrada/saída, a memória é o fator limitante. Especificamente para este cenário combinado, observa-se que o consumo de rede entre primário e *backup* foi semelhante para todos os intervalos de *checkpoints* (Figura 11).

3.6 Cenário B: transferência de dados via rede

Este experimento avalia a latência percebida por um cliente conectado a uma MV protegida por Remus. Foi usada a ferramenta Netcat[3] para minimizar a sobrecarga de aplicação que pudesse influenciar nos resultados, já que o Netcat apenas envia os dados por um *socket* TCP, sem um protocolo de aplicação. Medimos o tempo necessário para cada transferência de 50 MB entre o cliente e a MV, sob diferentes intervalos de *checkpoints*. Testes de *upload* representam transferências do cliente para a MV, enquanto *download* representa o sentido oposto. Desta forma, caracterizamos o impacto causado pelo *buffer* de rede e pela execução da MV em modo especulativo.

[2] Disponível em https://www.isc.org/downloads/bind/

[3] Disponível em http://netcat.sourceforge.net

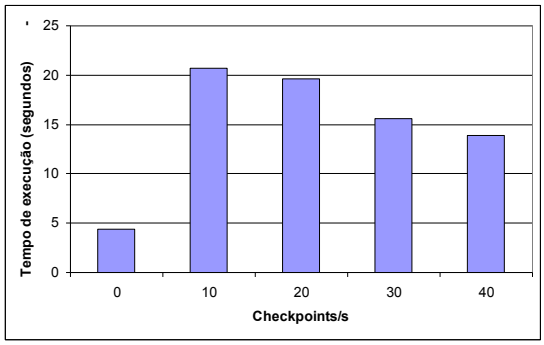


Figura 12. Tempo de transferência de um arquivo de 50 MB do cliente para a MV (*upload*).

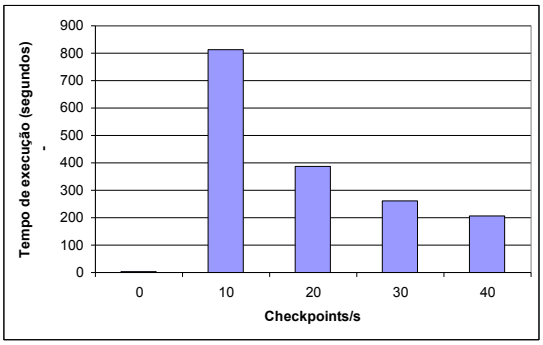


Figura 13. Tempo de transferência de um arquivo de 50 MB da MV para o cliente (*download*).

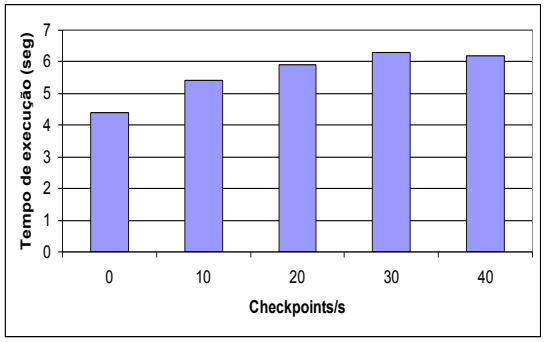


Figura 14. Tempo de transferência de um arquivo de 50 MB para a MV (*upload*), sem o *buffer* de rede ativo.

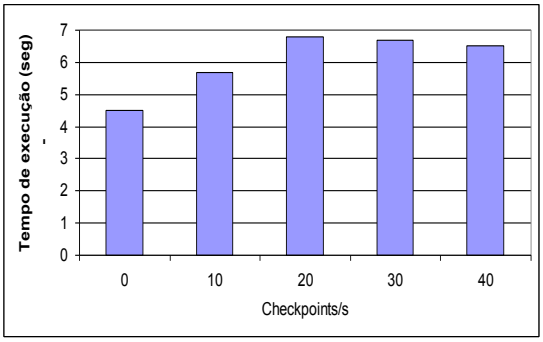


Figura 15. Tempo de transferência de um arquivo de 50 MB partindo da MV para o hospedeiro cliente (*download*), sem o *buffer* de rede ativo.

Ao contrário do que mostram os testes anteriores (cenário A), onde não há percepção de latência, a sobrecarga maior sobre a MV se dá com maiores intervalos de *cps/s*. A Figura 12 mostra que no melhor resultado, 40 *cps/s*, a sobrecarga é de 215% em relação à mesma transferência para uma MV sem proteção. No segundo caso, onde a transferência parte da MV (Figura 13), a menor sobrecarga é de aproximadamente 4500% (4,5 s sem Remus, contra 202 s com 40 *cps*/s). No pior caso, utilizando uma replicação de 10 *cps/s*, a sobrecarga está próxima a 18.000%. Novamente, quanto mais *cps/s*, menor a latência percebida pelo cliente. Porém, mesmo no melhor caso, o impacto no desempenho de *download* é acentuado, podendo inviabilizar o uso do Remus.

A diferença da latência percebida entre os dois testes (*upload* e *download*) pode ser explicada pela atuação do *buffer* de rede. A MV, mesmo em modo especulativo, nunca deixa de receber requisições. O que há é um bloqueio às respostas das requisições até que o período de *checkpoint* esteja completo. Não haverá transmissão de dados partindo da MV enquanto o hospedeiro *backup* não devolver uma confirmação de recebimento completo relativo ao *checkpoint* em questão.

Para fins de análise de desempenho, Remus permite a execução sem o *buffer* de rede ativo (o que reduz o nível de proteção). As Figuras 14 e 15 mostram o resultado para transferências de 50 MB entre o cliente e a MV, tanto para *upload* quanto para

download, respectivamente, sem a utilização do *buffer* de rede. Enquanto com *buffer* de rede ativo o tempo para transferência de um arquivo de 50 MB para a MV foi de 13,9 s, sem o *buffer* foi de 6,2 s com 40 *cps/s*. No caso de *download*, também a 40 *cps/s*, o tempo necessário é 202 s, contra 6,5 s sem o *buffer*. Neste último caso, pode-se afirmar que o *buffer* de rede foi responsável em aumentar a latência percebida pelo cliente em mais de 3.000%. Outra característica dos testes sem *buffer* de rede ativo é que o melhor desempenho na execução das MVs se dá com intervalos maiores entre *checkpoints*, a exemplo do que ocorre com aplicações não sensíveis a latência (cenário A). Há de se ressaltar que esta é uma opção apenas para *benchmark,* já que o *buffer* de rede é essencial para que o estado das MVs esteja sempre igual na visão do cliente em caso de ocorrência de faltas.

Em suma, este experimento indica que o *buffer* de rede é o principal responsável pela latência percebida pelo cliente quando o Remus está em execução. O tempo necessário para transferir o *buffer* local para o hospedeiro *backup*, somado ao tempo em que a MV fica suspensa até que os dados sejam transferidos para o *buffer* local, são as principais causas. Independente do intervalo de *cps*, a latência percebida pelo cliente é sempre alta, principalmente quando é a MV a origem das transferências. *Checkpoints* mais frequentes, como 40 *cps/s*, fazem com que o *buffer* a ser transferido entre primário e *backup* seja menor, o que leva a uma resposta mais rápida às requisições solicitadas pelos clientes. Porém, frequências de *checkpointing* menores resultam em menos suspensões, deixando a MV livre para processamento por mais tempo (conforme discutido nos experimentos do cenário A).

3.7 Cenário B: *Benchmark* ab

Este experimento avalia o desempenho de um servidor web Apache[4] versão 2.2.22 (com a configuração padrão do OpenSUSE) executando na MV, atendendo a 100 conexões simultâneas durante 60 segundos. O arquivo a ser lido constantemente pelo cliente compreende uma página de 72950 bytes (phpinfo.php).

A Figura 16 mostra o percentual das requisições respondidas por segundo. Com 5,645ms, 50% das requisições sob 40 *cps/s* já haviam sido respondidas, contra 14,428ms para a mesma porcentagem sob 10 *cps/s*. Em relação ao número de requisições atendidas, a diferença entre o intervalo de *checkpoints* é ainda maior. No pior caso, com 10 *cps/s*, o servidor web respondeu a apenas 53 requisições durante o período. No melhor caso, com 40 *cps/s*, 200 requisições foram atendidas. Porém, se comparado ao teste sem Remus, no qual o servidor respondeu a 9549 requisições durante os mesmos 60s, houve uma sobrecarga no desempenho superior a 4.600% conforme mostra a Figura 17. A média da vazão de dados entre o primário e o *backup* (Figura 19) variou pouco em relação a 40 ou 30 *cps/s*. Em suma, os resultados mostram que quanto menor o intervalo de *checkpoints*, melhor o desempenho, menor a latência e maior o fluxo de rede utilizado para a replicação.

[4] Disponível em http://www.apache.org.

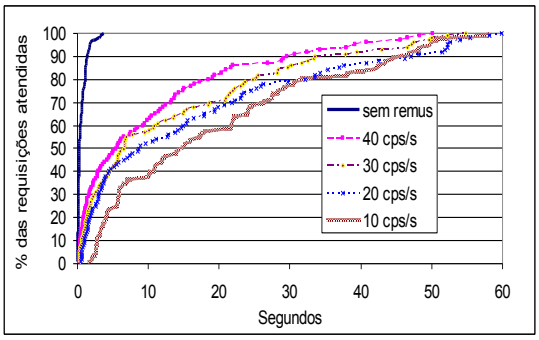


Figura 16. Percentual das requisições atendidas em 60 segundos.

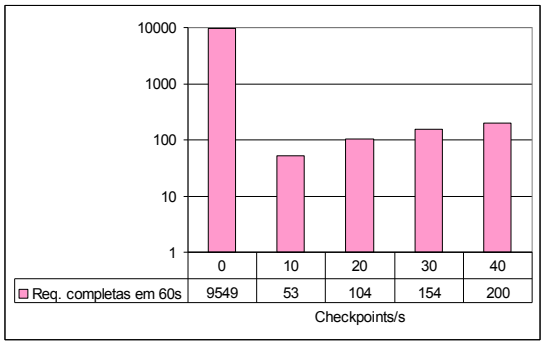


Figura 17. Número de requisições atendidas em 60s sob diferentes *cps/s.*

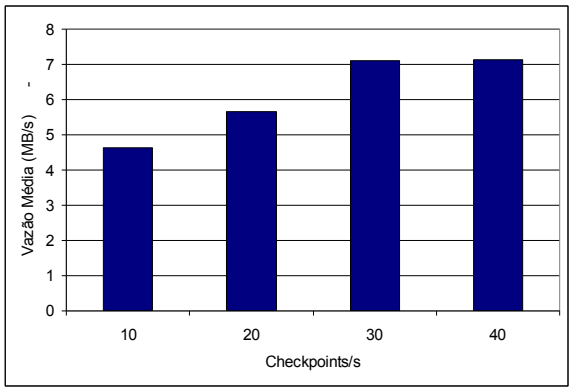


Figura 18. Média da vazão de dados entre primário e *backup* durante a execução do *benchmark* ab.

Nas operações em que não há uma percepção de latência, como no caso da compilação, a média de tráfego entre primário e *backup* se mantém constante, mesmo com diferentes intervalos de *checkpoints*. Em relação ao *benchmark ab*, onde há percepção de latência, tal característica não se repete.

3.8 Cenário B: *Benchmark* tbench

Este experimento analisa o comportamento de um servidor de arquivos replicado com Remus. Para tal, o *benchmark* tbench[5] simula as operações comumente executadas em um servidor de arquivos SMB (*Server Message Block*). Este *benchmark* simula somente a percepção do usuário, ou seja, somente as mensagens enviadas sem executar operações de leitura e escrita em disco. Neste teste foram utilizadas as configurações padrões sugeridas na documentação do tbench. A Tabela I apresenta o número de operações executadas e a latência observada pelo usuário do serviço.

[5] Disponível em http://linux.die.net/man/1/tbench e http://dbench.samba.org.

Operação	Sem Remus		40 *cps/s*		30 *cps/s*		20 *cps/s*		10 *cps/s*	
	ops	lat (ms)	ops	lat (ms)	ops	lat (ms)	ops	lat (ms)	ops	lat (ms)
Close	181968	3,08	5755	26,36	4105	35,19	2404	52,09	2447	101,83
Rename	10504	3,25	293	26,83	220	33,07	122	51,11	139	101,54
Unlink	50139	3,08	1368	25,91	1062	35,52	355	51,01	321	101,24
Find	86873	3,34	2606	51,44	1936	56,56	1188	84,45	944	133,18
Write	122390	5,55	3742	45,46	2724	59,54	1643	83,18	1713	151,54
Read	388191	6,13	11525	438,11	8891	575,19	5975	889,59	3536	1309,33
Flush	17376	3,57	491	25,93	370	37,11	150	50,82	298	102,05

Tabela I – Número de operações executadas em um servidor SMB durante 600 segundos e a média da latência percebida pelo cliente para cada operação.

Assim como no experimento da Seção 3.7, a latência aumenta conforme aumenta o intervalo entre *checkpoints*. A Figura 19 mostra a média de latência para cada operação.

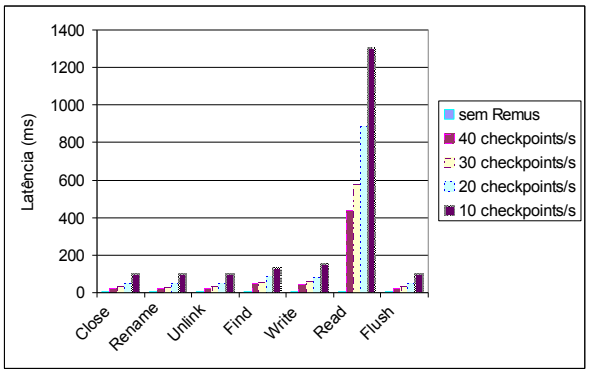


Figura 19. A latência média percebida pelo cliente para cada operação de rede em um servidor de arquivos.

4. Trabalhos relacionados

A migração e replicação de máquinas virtuais são temas promissores, já abordados em outros contextos. Enquanto algumas soluções preocupam-se com ambiente LAN, outras são voltadas para WAN, onde a instabilidade e alta latência são grandes desafios. Além disso, nem todas abordagens objetivam alta disponibilidade: o processo de migração pode ser realizado por motivos de desempenho ou simplesmente manutenção de um hospedeiro físico. Por exemplo, o mecanismo nativo de *live migration* [Clark et. al, 2005] implementado pelo hipervisor Xen atua na migração de uma MV em uma rede local. O conteúdo da memória e o estado da CPU virtual são transferidos para outro hospedeiro sem que haja perda de conexão entre cliente e serviços hospedados na MV, ou seja, não ocorre um alto intervalo de indisponibilidade. Como não há replicação do disco, o mesmo deve ser compartilhado ou copiado para o hospedeiro físico de destino antes da migração. CloudNet [Wood et al, 2011] propõe a migração de MVs sobre uma WAN buscando um balanceamento de carga. A tomada de decisão para tal migração é motivada pela latência observada pelo cliente final e/ou limites de capacidade. A replicação de disco fica a cargo do DRBD [Reisner e Ellenberg, 2005].

O mecanismo proposto pelo SecondSite [Rajagopalan et. al, 2012] utiliza as mesmas técnicas de replicação do Remus, como migração assíncrona e *checkpoints*, diferenciando o ambiente alvo: SecondSite propõe a replicação completa de um *site* entre *datacenters* geograficamente distribuídos. Para tal, utiliza um servidor externo para detecção de falhas, responsável por ativar o *site* secundário quando necessário. O objetivo é migrar as MVs em caso de catástrofes naturais, faltas de *hardware*, *links* ou energia, para outro *datacenter*. Nesta solução, a manutenção da conectividade das MVs é obtida através de configurações de roteamento previamente informadas nos roteadores BGP sob um único Sistema Autonômo. Assim como Remus, SecondSite utiliza o DRBD para replicar o disco, de uma forma assíncrona.

Considerando a implementação destes mecanismos, existe uma crescente busca em relação ao desempenho final. Por exemplo, *live migration* migra apenas o que foi alterado na memória em relação à rodada anterior. Este processo, conhecido como cópia iterativa, inicia-se logo após uma cópia inteira da memória ser transmitida para o hospedeiro *backup*. Se na fase *stop-and-copy* ainda restar um número grande de dados na memória a serem copiados, a MV poderá ficar indisponível por um longo período. SecondSite e Remus, além da mesma técnica utilizada pelo *live migration* (replicar a cada *checkpoint* apenas o que foi alterado na memória em relação ao *checkpoint* anterior), também utilizam outras duas técnicas para diminuir a sobrecarga causada pela constante replicação: compressão do tamanho do *checkpoint* e utilização de um *buffer* local para armazenar o estado da MV antes de transferi-lo. CloudNet, por sua vez, utiliza *live migration* para migrar uma MV em um ambiente WAN. Para diminuir, ou otimizar, o número de rodadas necessárias para a transferência, localiza dados repetidos na memória e no disco, transferindo-os apenas uma vez para o hospedeiro de destino. Os estudos para diminuir a indisponibilidade e a sobrecarga estão sempre voltados à quantidade de dados transferida do hospedeiro primário para o *backup*, mas na replicação há mais um fator relevante, que é a frequência com que as cópias ocorrem. Remus replica a MV inteira para o hospedeiro *backup* a uma taxa de vários *cps/s*. Como demonstrado na Seção 3, a frequência de *checkpoints* influencia no desempenho da MV, já que cada *checkpoint* representa um período de indisponibilidade.

Especificamente considerando o desempenho das aplicações hospedadas nestas arquiteturas, [Cully et al, 2008] aborda um processo de compilação como *benchmark* para avaliar o desempenho da MV em relação ao número de *cps/s*, porém não há estudos isolados em relação a CPU, memória e E/S. Ainda, os autores simularam a percepção de um usuário através do *benchmark* SPECweb, não considerando aplicações interativas como abordado em nossa análise experimental. [Magalhães et al, 2011] analisaram o desempenho de dois tipos de migração de MVs hospedadas em Xen: *stop-and-copy* e pré-cópia. Em sua análise, observaram que o tempo de indisponibilidade é diretamente relacionado com o tamanho da memória ocupada pela MV e com o tipo da aplicação em execução. Esta situação é semelhante no contexto de nosso trabalho dada a proximidade existente entre o Remus e o mecanismo de *live migration*.

5. Conclusão

A análise dos resultados mostra que o mecanismo de replicação Remus induz uma sobrecarga não desprezível quanto ao desempenho das aplicações hospedadas nas MVs protegidas. Em relação a CPU, memória e disco é preferível um intervalo maior entre

checkpoints, ou seja, menos *checkpoints* por segundo, porém o oposto pode ser dito em relação a aplicações sensíveis a latência, como um servidor web ou de arquivos. Sobretudo, aplicações com uso intensivo de operações em memória tendem a sofrer um maior impacto devido à natureza do mecanismo de replicação. A vazão média de replicação entre primário e *backup* varia conforme a natureza das operações executadas na MV e a freqüência de *checkpoints*. Quanto maior o intervalo entre *checkpoints*, menor a vazão média, pois o processo de compressão do *checkpoint* opera com *buffers* maiores, em menor quantidade, diminuindo o volume de dados que são transferidos.

Interromper uma MV frequentemente é o principal responsável pela sobrecarga identificada nas aplicações que não respondem a requisições externas. Por outro lado, nas aplicações onde há uma percepção de latência, além dos *checkpoints*, há também o fato da MV estar constantemente em modo de execução especulativo, onde as requisições somente são respondidas e entregues ao requisitante após o hospedeiro *backup* acolher completamente o *checkpoint* anterior. Nesse cenário, observa-se que a replicação de MVs utilizando o mecanismo de proteção Remus pode comprometer o desempenho final da aplicação hospedada.

Referências

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. "Xen and the art of virtualization". Proc. of the 19th ACM Symp. on Operating Systems Principles (SOSP), 2003.

Bressoud, T. C., and Schneider, F. B. "Hypervisor-based fault tolerance". *ACM Trans. on Computer Systems*, 14(1), 80-107, 1996.

Budhiraja, N., Marzullo, K., Schneider, F. B., and Toueg, S. "The primary-backup approach". In *Distributed Systems (2nd Ed.)*, Cap. 8. Sape Mullender (Ed.), ACM Press, 1993.

Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A. "Live migration of virtual machines". Proc. of the 2nd USENIX Symp. on Networked Systems Design and Implementation (NSDI), 2005.

Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N., and Warfield, A. "Remus: High availability via asynchronous virtual machine replication". Proc. of the 5th USENIX Symp. on Networked Systems Design and Implementation (NSDI), 2008.

Magalhães, D. M. V.; Soares, J. M.; Gomes, D. G. "Análise do Impacto de Migração de Máquinas Virtuais em Ambiente Computacional Virtualizado". Anais do XXIX SBRC, 2011.

Pearce, M., Zeadally, Sherali., and Hunt, R. "Virtualization: Issues, security threats, and solutions". *ACM Computing Surveys*, 45(2), 2013.

Rajagopalan, S., Cully, B., O'Connor, R., and Warfield, A. "Secondsite: Disaster tolerance as a service". Proc. of the 8th ACM SIGPLAN/SIGOPS Conf. on Virtual Execution Environments (VEE), 2012.

Reisner, P., and Ellenberg, L. "DRBD v8 – replicated storage with shared disk semantics". Proc. of the 12th International Linux System Technology Conference, 2005.

Stevens, W. R. *TCP/Illustrated, Vol. 1: The Protocols*. Addison-Wesley, 1994.

Wood, T., Ramakrishnan, K. K., Shenoy, P., and van der Merwe, J. "Cloudnet: Dynamic pooling of cloud resources by live WAN migration of virtual machines". Proc. of the 7th ACM SIGPLAN/SIGOPS Conf. on Virtual Execution Environments (VEE), 2011.

Provedores de Identidade Resilientes e Confiáveis

Diego Kreutz[1], Eduardo Feitosa[2], Hugo Cunha[2]

[1]LaSIGE/FCUL, Lisboa, Portugal

[2]IComp/UFAM, Manaus, Brasil

kreutz@computer.org, {efeitosa, hugo.cunha}@icomp.ufam.edu.br

Resumo. *Este artigo apresenta o desenvolvimento de uma arquitetura para provedores OpenID resilientes e seguros, cujo objetivo é garantir propriedades essenciais como integridade, alta disponibilidade e confidencialidade de dados sensitivos. Para este fim são utilizados algoritmos para tolerar falhas arbitrárias, é proposto um componente seguro, com uma interface bem definida, e é detalhada a arquitetura que provê as propriedades almejadas ao sistema. A solução proposta supera trabalhos similares em diferentes aspectos, como vazão, latência, tolerância a falhas arbitrárias e confidencialidade (sem comprometer a escalabilidade do sistema). Os resultados demonstram que uma única instância do sistema suporta demandas de ambientes, como infraestruturas de rede e sistemas Web, com mais de 200k usuários.*

1. Introdução e Motivação

Provedores de identidade (IdPs) são empregados como fontes primárias de autenticação e/ou autorização em diferentes tipos de serviços e sistemas. Essencialmente, sistemas Web vem utilizando, em larga escala, provedores de identidade (externos) baseados em OpenID [Recordon and Reed 2006]. Isso significa que o usuário pode ter uma única identidade, em um provedor a sua escolha, para acessar serviços e sistemas de diferentes domínios. Adicionalmente, a utilização de IdPs também vem promovendo a criação de sistemas com suporte a Single-Sign-On (SSO), ou seja, o usuário autentica-se em um único serviço e passa automaticamente a ter acesso a outros serviços, os quais aceitam a mesma sessão de autenticação.

Como consequência, diferentes serviços e sistemas passam a confiar na autenticação e autorização de IdPs. No entanto, apesar da rápida expansão em termos de utilização, ainda persistem diferentes problemas relacionados com a disponibilidade e confiabilidade de provedores de identidade [Uruena et al. 2012, Kreutz et al. 2013b, Kreutz et al. 2013a, Kreutz et al. 2014]. Em termos práticos, a maioria dos IdPs não contempla aspectos de segurança e propriedades como confidencialidade, integridade e disponibilidade.

Existem alguns trabalhos que objetivam melhorar a capacidade de tolerar falhas e intrusões em serviços como Kerberos v5 [de Sousa et al. 2010], RADIUS [Malichevskyy et al. 2012] e provedores OpenID [Barreto et al. 2013], denominado aqui de OpenID-VR. Somente este último prove tolerância a intrusões em provedores OpenID. Entretanto, nenhum desses trabalhos explora como mitigar os diferentes tipos de falhas e ataques ou questões como componentes seguro escaláveis para provedores OpenID. O OpenID-VR tolera apenas intrusões relativas ao protocolo OpenID propriamente dito. Nesse sentido, a proposta aqui apresentada,

denominada de OpenID-PR, detalha uma arquitetura resiliente e confiável para IdPs, capaz de tolerar quaisquer falhas arbitrárias (não apenas do protocolo OpenID), prover propriedades como a confidencialidade de dados sensitivos e atender demandas de ambientes reais.

Os principais objetivos do trabalho são: (1) Identificar, analisar e discutir os principais blocos de construção para melhorar a resiliência e confiabilidade de IdPs OpenID, tomando como base os modelos e componentes anteriormente propostos [Kreutz et al. 2013b, Kreutz et al. 2013a, Kreutz et al. 2013c]; (2) suportar falhas arbitrárias em provedores OpenID, ou seja, não restringir-se ao protocolo OpenID; (3) apresentar e discutir os mecanismos para suportar falhas arbitrárias e intrusões em IdPs; (4) manter a compatibilidade com as infraestruturas OpenID existentes; (5) analisar o desempenho da solução em diferentes cenários, como uma única máquina física, múltiplas máquinas em um único data center e múltiplas máquinas em múltiplos data centers.

Este artigo está organizado da seguinte forma: a Seção 2 apresenta alguns conceitos básicos e contextualiza o trabalho no âmbito do projeto SecFuNet [Kreutz et al. 2013b]. A Seção 3 descreve a arquitetura e os elementos de projeto do OpenID resiliente. Em seguida, as Seções 4 e 5 detalham a implementação e discutem os resultados experimentais da arquitetura proposta. Por fim, a Seção 6 apresenta as considerações finais.

2. Contexto e Trabalhos Relacionados

O trabalho insere-se no contexto do projeto SecFuNet, mais especificamente na parte de gestão de identidades através de provedores OpenID. Para assegurar a confidencialidade dos dados sensitivos, num servidor OpenID, são necessários elementos como componentes seguros. Problemas similares tem sido abordados em outros serviços, como Kerberous v5 e RADIUS, bem como com outras abordagens, como "replicação virtual", em provedores OpenID.

Gestão de Identidade no contexto do SecFuNet. Um dos pressupostos do projeto SecFuNet é a utilização de componentes seguros no processo de autenticação de usuários e sistemas. Entretanto, a maioria desses componentes, baseados em hardware, possui limitações de escalabilidade. Por exemplo, os smart cards disponíveis no mercado não podem ser comparados com unidades de processamento e armazenamento como CPUs x86 e discos SSD. Consequentemente, um dos desafios é projetar um sistema de identificação e autenticação mais seguro e disponível sem comprometer questões de escala, disponibilidade, integridade e confidencialidade.

A solução proposta para a implantação de IdPs mais confiáveis e resilientes baseia-se no OpenID para o estabelecimento de relações de confiança entre usuários, IdPs e provedores de serviços. Além disso, propõe o uso de componentes seguros, especificamente projetados para o OpenID, para isolar e proteger a confidencialidade de dados sensitivos, como chaves privadas e asserções de autenticação.

Componentes Seguros. Componentes seguros podem ser implementados utilizando diferentes técnicas e tecnologias como o isolamento provido por *hypervisors* (e.g. máquinas virtuais isoladas, componentes de software no domínio do *hypervisor*), Trusted Platform Modules (TPMs), arquiteturas como Secure Virtual Machine (SVM) da AMD, tecnologias como Trusted Execution Technology (TXT) da Intel,

e soluções baseadas em FPGAs [Niedermayer et al. 2014, Viejo 2014].

A solução proposta considera duas formas de prover componentes seguros: (1) através do isolamento provido por *hypervisors* ; e (2) por meio de conceitos e premissas de TPMs, que pode ser aplicado a sistemas de autenticação para garantir a confidencialidade de dados sensitivos, sem comprometer a escalabilidade do sistema.

Trabalhos Relacionados. Até o presente momento, a tolerância a falhas e intrusões em IdPs tem sido pouco explorada na literatura. Os três trabalhos descritos a seguir são confrontados e comparados com a presente proposta.

Os dois primeiros exploram técnicas de tornar os serviços Kerberos v5 (denominado de Typhon) [de Sousa et al. 2010] e RADIUS [Malichevskyy et al. 2012] mais resilientes. Ambos utilizam técnicas e algoritmos tradicionais de replicação de máquinas de estado para tolerar falhas arbitrárias nos respectivos serviços. Adicionalmente, propõe componentes seguros, pouco escaláveis (se aplicados a componentes de hardware), para armazenar os dados sensíveis, como chaves secretas e tokens de sessão, e garantir a confidencialidade do sistema em caso de intrusão nas réplicas.

O terceiro trabalho propõe um IdP capaz de tolerar intrusões de operação do protocolo OpenID [Barreto et al. 2013]. O OpenID-VR utiliza uma abordagem baseada em "replicação virtual". Em outras palavras, o sistema assume que o *hypervisor* forma uma base de computação confiável. Sendo assim, as réplicas (máquinas virtuais sobre um mesmo *hypervisor*) utilizam uma área de memória compartilhada para executar protocolos de consenso, por exemplo. O pressuposto é que essa estratégia reduz a sobrecarga de comunicação imposta por abordagens baseadas em troca de mensagens (comunicação via rede), como é o caso do Typhon e do RADIUS resilientes. Além disso, o OpenID-VR utiliza um único serviço de autenticação, baseado em processadores seguros (grids de smart cards) para a autenticação dos usuários.

Em resumo, a abordagem do OpenID-VR possui pelo menos dois desafios: (1) a escalabilidade do sistema, uma vez que grids de smart cards geram uma sobrecarga significativa ao processo de autenticação, que pode ultrapassar a 2 segundos [Urien and Dandjinou 2006], além de impactar significativamente na vazão do sistema; (2) tolerar apenas instruções relativas ao protocolo OpenID, ou seja, falhas de infraestruturas (físicas e/ou lógicas) não são suportadas. Além disso, o serviço de autenticação é um ponto único de falha.

A Tabela 1 resume as principais características e componentes dos três sistemas, bem como da solução proposta. Nota-se que a solução OpenID-VR, por basear-se em máquinas virtuais e uma única máquina física, não é capaz de tolerar falhas físicas (e.g. quedas de energia, defeitos em discos rígidos, perdas de conectividade, entre outros), falhas lógicas (e.g. falhas de software, falhas de configuração de rede e/ou conectividade, entre outros) e nem ataques de exaustão de recursos, como os reportados em [Kreutz et al. 2013b]. Adicionalmente, uma falha do *hypervisor* ou do serviço de autenticação seguro compromete toda a operação do serviço OpenID, uma vez que ambos são únicos. Por outro lado, a solução proposta permite utilizar de 1 a $3f + 1$ componentes seguros (*# c. seguros*). Além disso, propõe um componente seguro para o desenvolvimento de subsistemas de autenticação escaláveis (*C. s. esc.*). Finalmente, a vazão do sistema mostra-se superior às demais soluções.

Resumidamente, os principais diferenciais da solução proposta, quando com-

Tabela 1. Comparação dos trabalhos relacionados e abordagem proposta

Sistema	Falhas fís.	Falhas lóg.	Tipos de ataques	# réplicas	# c. seguros	C. s. esc.	Vaz.	Lat.
Typhon	sim	sim	quaisquer(*)	$3f+1$	$3f+1$	não	280	5.7ms
RADIUS	sim	sim	quaisquer(*)	$3f+1$	$3f+1$	não	100	200ms
OpenID-VR	não	sim/não	OpenID	$2f+1$	1	não	—	7ms
OpenID-PR	**sim**	**sim**	**quaisquer(*)**	$3f+1$	1 a $3f+1$	**sim**	**2440**	**7ms**

(*) Estes sistemas toleram potencialmente qualquer tipo de falha ou ataque em até f réplicas do sistema.

parada a solução OpenID-VR, são: (1) maior disponibilidade, resistindo a diferentes tipos de falhas físicas e lógicas; (2) resistência a ataques de exaustão de recursos, uma vez que as réplicas podem ser instanciadas em diferentes máquinas físicas e/ou domínios administrativos; (3) potencial suporte a quaisquer tipos de ataques e não apenas a ataques ao protocolo OpenID; (4) maior robustez no subsistema de autenticação, uma vez que pode ser composto por 1 a $3f + 1$ (até um por réplica) componentes seguros; (5) latência de autenticação equiparável, apesar de utilizar protocolos de troca de mensagens ao invés de memória compartilhada; (6) suporte a componentes seguros do tipo TPM, permitindo escalabilidade do subsistema de autenticação ao armazenar os dados sensíveis (e.g. chaves de sessão, dados de usuários, etc.) nas réplicas de forma protegida (i.e., cifrada); e (7) capacidade de tirar vantagem de ambientes multi-data center e/ou multi-cloud, usufruindo dos respectivos mecanismos de defesa dessas infraestruturas, como mecanismos de mitigação de ataques de negação de serviços [Prince 2013, Prince 2012].

3. Arquitetura do Sistema

A arquitetura do sistema proposto recorre a conceitos, técnicas e mecanismos que permitem construir sistemas baseados no conceito de segurança automática [Verissimo et al. 2006]. Em outras palavras, não corrige todos os problemas de segurança de IdPs, mas sim utiliza técnicas para tolerar falhas arbitrárias e intrusões.

3.1. Visão geral da arquitetura

As configurações arquiteturais ilustradas nas Figuras 1 e 2 são baseadas (e representam uma extensão) em um modelo funcional para serviços de autenticação e autorização, anteriormente proposto [Kreutz et al. 2013a]. Essencialmente, são cinco elementos principais: (a) cliente; (b) serviço e *relying party*; (c) gateway OpenID; (d) réplicas OpenID; e (e) componentes seguros. O conjunto dos gateways e réplicas OpenID, juntamente com o(s) componente(s) seguro(s), formam a infraestrutura do IdP. A função essencial do(s) componente(s) seguro(s) é garantir a confidencialidade dos dados sensitivos, como chaves privadas, certificados, chaves de sessão e asserções OpenID. Além disso, todas as operações que requerem acesso aos dados sensitivos são executadas pelo(s) componente(s) seguro(s).

O cliente representa um usuário ou dispositivo (e.g. máquina virtual) que quer acessar um serviço. Este serviço (que permite autenticação via OpenID) é vinculado a uma *relying party*, a qual redireciona o cliente para o seu próprio IdP.

O gateway é um elemento transparente para o usuário final e *relying party*, pois ele representa o servidor OpenID. As requisições que chegam ao gateway são simplesmente repassadas às réplicas OpenID. A função do gateway é mascarar a

replicação ativa, i.e., um pedido de um cliente é enviado para $3f_R + 1$ réplicas do sistema. O gateway mantém a compatibilidade com sistemas OpenID 2.0 existentes.

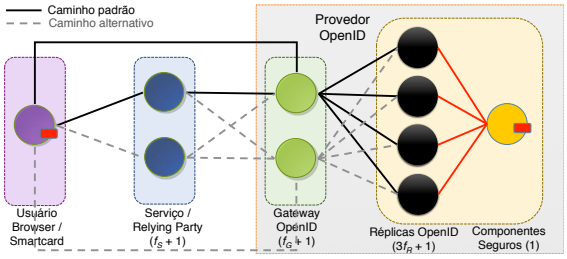


Figura 1. 1 **componente seguro**

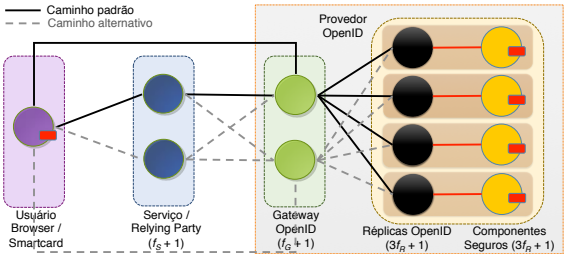


Figura 2. $3f_R + 1$ **comp. seguros**

O IdP propriamente dito é considerado a parte crítica do modelo funcional, pois é ele quem identifica e autentica usuários para os serviços. Portanto, requer níveis de segurança e disponibilidade mais elevados para assegurar que usuários inválidos e/ou sem permissão não sejam aceitos nos serviços. Um provedor de identidade pode ser parte integrante de um domínio local, bem como um serviço prestado por terceiros. Na prática, o caso mais comum tem sido os diferentes serviços aceitarem autenticações de usuários de IdPs externos. Devido às consequências que podem ser geradas por eventuais IdPs falhos ou comprometidos, é assumido que IdPs OpenID devem tolerar falhas arbitrárias, que podem ser causadas por comportamentos inesperados da infraestrutura ou do sistema, bem como por ataques. Adicionalmente, o IdP OpenID resiliente deve tolerar também intrusões. Portanto, são necessários componentes seguros para resguardar a confidencialidade de dados sensitivos.

Outro aspecto a observar nas Figuras 1 e 2 é o número de componentes seguros. No primeiro caso, as réplicas OpenID utilizam um único componente seguro, de forma compartilhada. Em termos práticos, o componente seguro irá responder à réplica solicitante somente após receber o mesmo pedido de $3f_R + 1 - f_R$ réplicas. Portanto, mesmo com um único componente seguro, não basta a um atacante comprometer apenas f_R réplicas para conseguir fazer com que o componente seguro gere pedidos válidos a partir de dados de entrada forjados ou modificados. Este primeiro caso é similar a solução OpenID-VP, ou seja, as réplicas e o componente seguro podem estar a executar sobre uma mesma plataforma de hardware e *hypervisor*, por razões de desempenho (menor latência de comunicação e potencial maior vazão).

Por outro lado, a configuração apresentada na Figura 2 visa, além de manter a integridade e a confidencialidade, também a alta disponibilidade do sistema. Esta configuração é menos suscetível a ataques de exaustão de recursos, falhas físicas e lógicas, além de outros tipos de ataques, como DDoS de grandes proporções. Isso deve-se ao fato de esta configuração do sistema poder tirar proveito dos mecanismos de segurança e proteção de diferentes máquinas físicas e/ou ambientes administrativos. Além disso, essa segunda configuração pode ainda ser utilizada para aumentar o desempenho do sistema, em especial a capacidade de processar requisições de autenticação por unidade de tempo.

É importante ressaltar ainda que a pilha de protocolos não muda. As comunicações com o serviço OpenID continuam a ser via HTTP e/ou SSL (ou TLS para autenticação mútua). O que ocorre é que o gateway recebe as requisições

SSL/HTTP e as encapsula no protocolo de tolerância a falhas Bizantinas (BFT). Consequentemente, um IdP OpenID resiliente pode facilmente, e de forma transparente, substituir um IdP OpenID tradicional. O cliente, ou a *relying party*, não vai notar nenhuma diferença funcional ou operacional no sistema.

3.2. Tipos de falhas toleradas

A Figura 3 ilustra os mecanismos de detecção e/ou mascaramento de falhas entre os diferentes elementos. Para mitigar falhas, além de suportar falhas por parada, há mecanismos complementares de detecção de falhas entre o cliente, *relying party* e gateway. Exemplos incluem mensagens corrompidas e pacotes mal formados.

Entre o gateway e as réplicas existem mecanismos de replicação ativa, utilizando máquina de estados replicada, para mascarar falhas arbitrárias de até f_R réplicas. Em outras palavras, qualquer tipo de comportamento, como atrasos, respostas mal formadas e mensagens corrompidas, em até f_R réplicas, irá ser mascarado sem afetar a operação do sistema. Por exemplo, pacotes mal formados ou respostas divergentes, de uma réplica falha, serão simplesmente descartadas.

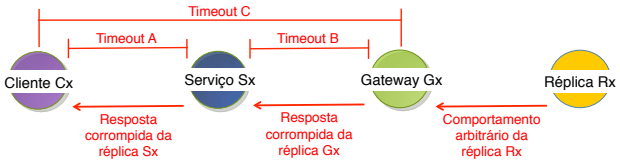


Figura 3. Detecção e mascaramento de falhas entre os elementos

3.3. Componentes seguros para provedores de identidade

O componente seguro deve ser mínimo e prover métodos como os discriminados na Tabela 2. Neste caso, as informações de associação e os *handlers* são armazenados no sistema de arquivos da réplica, contudo, cifrados e/ou assinados para assegurar a confidencialidade e integridade dos dados. Portanto, o componente seguro segue o modelo de comportamento de uma TPM, armazenando internamente informações mínimas necessárias para garantir algumas das propriedades do sistema.

Tabela 2. Interface do componente seguro

Método	Protocolos	Entrada	Saída
`VerifySignRSA`	TLS/SSL	Pacote com a assinatura a ser verificada.	Ack ou Nack de verificação da assinatura.
`SignRSA`	TLS/SSL	Dados a assinar.	Assinatura RSA para os dados de entrada, utilizando a chave (Pr_S).
`GenAssociation`	OpenID	Dois números inteiros e chave pública do cliente.	Informação de associação e a chave pública do serviço OpenID.
`GenerateNonce`	OpenID	—	Um *nonce* pseudo aleatório.
`SymmetricCipher`	TLS/SSL	Dados a cifrar.	Dados de entrada cifrados.

O componente seguro necessita armazenar somente quatro valores (K_{User}, Pr_S, Pu_{CA} e K_{Assoc}) para garantir as propriedades de integridade e confidencialidade do sub sistema de autenticação. Utilizando a chave K_{Assoc} e um *cipher* simétrico (`SymmetricCipher`), as informações de associação podem ser armazenadas, cifradas, na réplica sem comprometer a confidencialidade dos dados no caso de uma intrusão. Sem conhecer a chave secreta ou quebrar o algoritmo de cifra, um atacante não é capaz de ler as informações de associação do provedor de identidade.

O segredo K_{User} é utilizado para verificar a identidade e os dados do usuário. Cada entrada na tabela de usuários do sistema contém um MAC gerado com o segredo K_{User}. Portanto, um atacante não consegue gerar ou modificar entradas da tabela sem conhecer a chave secreta utilizada pelo componente seguro. Em outras palavras, apenas o componente seguro consegue gerar MACs válidos para as entradas da tabela de usuários. Alternativamente, um administrador do sistema também poderia conhecer a chave secreta para poder gerenciar (adicionar, modificar, remover) a tabela de usuários do sistema.

Já o certificado Pu_{CA} é necessário para verificar a identidade de um usuário através do respectivo certificado assinado pela CA. O sistema assume que usuários válidos são somente aqueles que possuem um certificado válido e assinado com a chave privada da CA. Portanto, a chave pública permite ao componente verificar a validade da identidade de um usuário.

A chave privada do serviço (Pr_S) é necessária para o caso de autenticação mútua, fim-a-fim, entre o cliente e o serviço OpenID, através de protocolos como o EAP/TLS. Esta chave é utilizada para verificar a assinatura (`VerifySignRSA`) da mensagem enviada pelo cliente e/ou *relying party*.

Finalmente, o método `GenerateNonce` tem como função criar um carimbo temporal e um número pseudo aleatório. O *nonce* tem como finalidade evitar ataques de *replay* junto às respostas de autenticação do OpenID.

4. Implementação

Para a implementação do serviço OpenID resiliente foi utilizada a biblioteca *openid4java* [OpenID4Java 2013] (versão 0.9.8), que suporta as versões 1.0 e 2.0 do OpenID. O protótipo desenvolvido utiliza a especificação 2.0 do OpenID.

A replicação ativa é realizada através da biblioteca BFT-SMaRt [Bessani et al. 2013]. Esta biblioteca prove um conjunto de módulos e protocolos de comunicação como o Mod-SMaRt, o VP-Consensus e Canais Confiáveis para comunicação entre as réplicas. Tanto Mod-SMaRt quanto VP-Consensus utilizam canais confiáveis em sua comunicação entre réplicas e componentes do sistema.

O componente seguro, que é um elemento independente do sistema, foi implementado como um módulo Java, com uma interface bem definida. Ele é utilizado pelas réplicas para executar algumas operações essenciais do protocolo, como detalhado na Figura 4. Adicionalmente, quando há um componente seguro por réplica, o Mod-SMaRt é utilizado também para comunicação entre os componentes seguros. Essa comunicação é necessária para chegar a um consenso entre os componentes seguros antes de responder à réplica. Os componentes seguros podem ser executados no mesmo ambiente das réplicas ou em máquinas virtuais isoladas.

Componentes do sistema. Tanto o cliente quanto a *relying party* podem ser aplicações (e.g. *browser*) ou serviços existentes. Para fins de avaliação experimental do protótipo, foram desenvolvidos um cliente e uma *relying party* em Java.

OpenID Gateway. É um elemento projetado para manter compatibilidade com infraestruturas OpenID existentes. Assim, recebe conexões TCP/IP (com HTTP e/ou HTTP sobre SSL) dos clientes e *relying parties* OpenID e simplesmente repassa os pacotes recebidos a interface BFT proxy do BFT-SMaRt. A biblioteca encarrega-se de enviar uma cópia dos pacotes para todas as réplicas. De maneira similar,

pacotes vindos das réplicas são enviados aos clientes e/ou *relying parties* através dos canais TCP/IP previamente estabelecidos.

Réplicas OpenID. A implementação atual do protótipo suporta OpenID 2.0 sobre HTTP. Uma réplica consiste em uma implementação do OpenID 2.0 a executar com o suporte da biblioteca BFT-SMaRt, que fornece os protocolos para replicação de máquina de estados. Com isso, todas as réplicas OpenID, corretas, atendem todas as requisições de autenticação na mesma ordem. Fazem parte dos trabalhos futuros o suporte a HTTPS e EAP-TLS para autenticação mútua entre: (a) cliente e *relying party*; (b) *relying party* e serviço OpenID; e (c) cliente e serviço OpenID. Vale a pena ressaltar que a implementação desse protocolos pode tirar partido da implementação EAP-TLS disponível no componente seguro do RADIUS [Malichevskyy et al. 2012].

Componente seguro. O componente seguro foi implementado em Java com a API de criptografia BouncyCastle [of the Bouncy Castle Inc. 2014]. O principal desafio de implementação foi o determinismo requerido pelas réplicas. Em outras palavras, os componentes seguros de cada réplica devem gerar dados de saída de maneira determinística. Para resolver esse problema foi utilizada a mesma solução da implementação do RADIUS resiliente, ou seja, uma adaptação da função de geração de números pseudo-aleratórios (PRF) do TLS [Malichevskyy et al. 2012].

4.1. Interação entre os elementos do sistema

A Figura 4 ilustra os passos de comunicação entre os diferentes elementos do OpenID resiliente. No passo 1, o usuário requisita acesso a um serviço através da *relying party*. No passo 2, a *relying party* apresenta ao usuário um formulário para ele inserir o seu identificador específico (URL de identificação, no passo 3). No passo 4, a *relying party* realiza a operação de descoberta em busca da informação necessária para continuar o processo de autenticação. A requisição de descoberta é encaminhada para o serviço OpenID replicado.

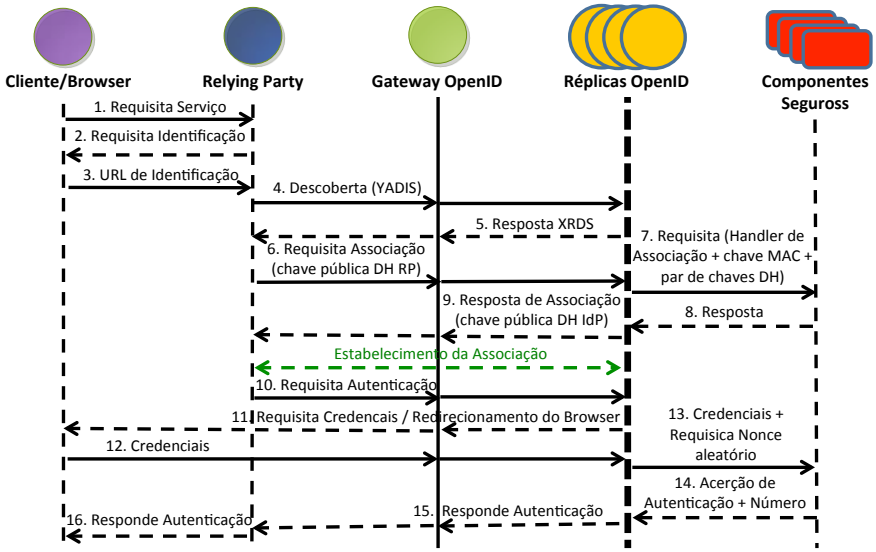


Figura 4. Visão geral das interações entre os componentes do sistema (OpenID 2.0)

O serviço OpenID responde à *relying party* (passo 5) com um documento XRDS que contém, entre outras coisas, uma lista de URLs (ponteiros para o provedor

OpenID) com as respectivas prioridades. Esta lista pode ser utilizada pela *relying party* nos próximos passos.

A seguir, a *relying party* tenta conectar-se com cada servidor OpenID (URL) contida na lista, de acordo com a ordem de prioridade pré-definida, e estabelecer uma associação protegida. A primeira conexão bem sucedida é utilizada para requisitar a associação (passo 6). Esta requisição contém dados como a respectiva URL e os dados Diffie-Hellman (DH) [Rescorla 1999] (chave pública, número primo, número gerador) da *relying party* para manter a associação segura.

No servidor OpenID, a requisição de associação é encaminhada ao componente seguro, que gerará o *handler* de associação e a chave MAC (passo 7). Na sequência, o servidor OpenID solicita ao componente seguro uma par de chaves DH, enviando os dados de entrada recebidos da *relying party* (número primo e número gerador). Depois disso, o componente seguro gera e retorna (passo 8) a informação requisitada (*handler* de associação, chave MAC e par de chaves DH do OpenID). A resposta de associação, contendo a chave pública DH do OpenID (entre outras informações), é enviada para a *relying party* (passo 9), completando a associação.

Assim que a associação entre a *relying party* e o servidor OpenID é estabelecida, a autenticação é iniciada. No passo 10, a *relying party* envia uma requisição de autenticação ao servidor OpenID, que é encaminhada ao browser do cliente como um pedido de credenciais (passo 11). O cliente envia as suas credenciais ao servidor OpenID (passo 12), que em seguida requisita a verificação das credenciais ao componente seguro. Adicionalmente, ele requisita também um *nonce* aleatório (passo 13). O componente seguro responde com a asserção de autenticação e o *nonce* gerado (passo 14). Depois disso, a resposta de autenticação é enviada a *relying party* (passo 15), que realiza as ações necessárias. Finalmente, a *relying party* envia uma resposta de autenticação ao cliente, completando o processo de autenticação (passo 16).

4.2. Configuração de elementos do sistema

O protótipo desenvolvido suporta duas configurações distintas, similares as apresentadas nas Figuras 1 e 2. Na primeira existe um único componente seguro, utilizado por todas as réplicas. Tanto as réplicas quanto o componente seguro podem estar a executar sobre uma mesma máquina física, em diferentes máquinas virtuais, por razões de desempenho. Neste caso, o componente seguro pode realizar uma votação majoritária simples, a partir dos pedidos das réplicas, sem precisar estabelecer comunicação com componentes seguros replicados.

A segunda configuração oferece uma plataforma mais robusta. Neste caso, há um componente seguro por réplica. Portanto, as réplicas podem ser instanciadas em uma única máquina física, em múltiplas máquinas físicas ou ainda em domínio distintos (e.g. múltiplos data centers). Nessa configuração até f_R componentes seguros podem falhar sem comprometer a operação do sistema.

Além disso, como o gateway e o OpenID replicado foram implementados em Java, ambos podem ser instanciados em diferentes sistemas operacionais, o que aumenta a robustez do sistema pelo fato de evitar vulnerabilidades comuns existentes num mesmo sistema. Outra característica interessante desta configuração é com relação à independência de *hypervisors*. Como o sistema baseia-se no paradigma de troca de mensagens, diferentes máquinas virtuais (réplicas) podem ser instanciadas

em distintos *hypervisors*, aumentando a robustez contra vulnerabilidades comuns tanto no sistema operacional quanto no sistema de virtualização.

5. Avaliação e Resultados Experimentais

Esta seção tem por objetivo discutir e avaliar a solução proposta em relação a diferentes ataques e tipos de falhas. Além disso, também são apresentadas avaliações de desempenho em três cenários distintos.

5.1. Ataques ao OpenID

O framework OpenID possui diferentes problemas de segurança, tanto de especificação quanto de implementação [Uruena et al. 2012, Kreutz et al. 2013b, Kreutz et al. 2013a]. A Tabela 3 resume algumas dos ataques conhecidos, bem como aponta as soluções adotadas no protótipo desenvolvido, como forma de mitigar os ataques.

Tabela 3. Análise dos Ataques ao OpenID

Ataque	Problema	Soluções
Man-in-the-middle	Interceptação de comunicações.	Segredo compartilhado entre RP e IdP, usando Diffie-Hellman, e protocolos como TLS para autenticação mútua entre usuário e servidor OpenID.
Negação de serviço	Potenciais ataques de DoS em RPs e IdPs.	(1) Limitar o número de requisições/s nos gateways (e.g. `iptables`). (2) No IdP, negar requisições com base nos valores das mensagens *openid.realm* e *openid.return_to*. (3) Utilizar mecanismos baseados em timeouts, verificação de mensagens corrompidas e pacotes mal formados.
Replay	No OpenID 2.0 é opcional a assinatura do *nonce*.	(1) Incluir o *nonce* na lista de informações assinadas. (2) Rejeitar *nonces* expirados.

5.2. Falhas arbitrárias e falhas por parada

O OpenID resiliente é comparado com uma versão livremente disponível e utilizada em IdPs OpenID, o JOIDS [openid-server Community 2010]. Este representa o caso comum de implementações OpenID disponíveis para criação de IdPs.

Falha por parada. Para provocar uma falha por parada é o suficiente matar o respectivo processo no sistema operacional. O OpenID resiliente não é afetado, i.e., continua a funcionar normalmente, sem perdas de desempenho, com falhas por parada de até f_R réplicas. Similarmente, falhas em até f_G gateways não afetam a operação do sistema, uma vez que as *relying parties* irão tentar estabelecer conexão com todos os gateways da lista. Por outro lado, em um sistema baseado no JOIDS, uma falha por parada interrompe o serviço.

Falhas arbitrárias. Servidores OpenID com o JOIDS não toleram falhas arbitrárias (e.g. bugs em diferentes camadas de software, problemas de configuração dos sistemas, atraso de mensagens intencionalmente provocadas por um atacante, e assim por diante). Como resultado, o serviço pode negar a autenticação de usuários legítimos e garantir asserções de associação a usuários não autorizados, por exemplo. Por outro lado, o OpenID resiliente tolera até f_R falhas arbitrárias nas réplicas OpenID. Isso inclui falhas de sistemas operacionais, falhas de bibliotecas e componentes de software, entre outras falhas físicas e lógicas, além de intrusões.

5.3. Impacto de falhas periódicas e ataques

Parando periodicamente até f_R réplicas. Para avaliar o sistema sobre circunstâncias adversárias, como falhas periódicas por parada, foi utilizado o ambiente

UFAM-VMs (detalhado na seção seguinte) com 20 e 40 clientes. Foi implementado um *script* para matar e re-iniciar uma das réplicas a cada 10 segundos durante a execução do sistema. Com periodicamente uma réplica a menos no sistema, houve um ligeiro aumento no desempenho do sistema, passando de 867.73 para 1009.86 autenticações/s com 20 clientes e de 984.59 para 1145.98 com 40 clientes. Isso é devido ao fato de menos réplicas no sistema ($3f_R + 1 - f_R$) gerar uma menor sobrecarga de comunicação nos protocolos utilizados para replicação de máquina de estados.

Ataque constante de DoS em até f_R réplicas. Foi utilizado o comando `hping3` para gerar um ataque de DoS constante, utilizando as flags SYN e ACK do TCP, na porta de comunicação de uma das réplicas. Esta réplica começou a ficar mais lenta e não receber todas as mensagens do gateway por causa do ataque. Consequentemente, as outras réplicas consideraram a réplica como comprometida e continuaram a manter o sistema em correta operação. Novamente, pode ser observado um pequeno aumento no desempenho do sistema, indo de 867.73 a 956.46 autenticações/s com 20 clientes, e de 984.59 para 1005.54 com 40 clientes.

5.4. Análise de Desempenho

Foram utilizados três ambientes distintos para avaliação do protótipo, conforme resumido na Tabela 4. Em cada ambiente foram instanciadas 5 máquinas virtuais (MVs), sendo uma para executar um gateway e os clientes e outras 4 para executar as réplicas e componentes seguros. No primeiro ambiente, UFAM-VMs, as máquinas virtuais (Debian Linux) foram suportadas pelo *hypervisor* Xen. O segundo ambiente, Amazon-EC2, é representado por nós de computação `m3.xlarge` [Amazon Web Services, Inc. 2014] no datacenter da Amazon de N. Virginia, rodando Ubuntu Server 13.10. O terceiro ambiente também utilizou nós de computação `m3.xlarge`, entretanto, distribuídos em três data centers da Amazon, N. Virginia (Ubuntu Server 13.10), N. Califórnia (Amazon Linux AMI) e Oregon (Ubuntu Server 12.04 LTS).

Tabela 4. Ambientes e configuração das máquinas virtuais

Environment	vCPUs	ECUs	MEM	Disk	Network
UFAM-VMs	2	—	2GB	20GB	Gigabit Ethernet
Amazon-EC2	4	13	15GB	2x40GB SSD	High Speed Gigabit
Amazon-DCs	4	13	15GB	2x40GB SSD	WAN pública

Vazão do sistema. A Tabela 5 resume os resultados para os três ambientes de teste. Foram utilizados 20, 40, 80 e 100 clientes simultâneos, executados na mesma máquina do gateway. Cada cliente foi configurado para realizar 2.000 requisições de autenticação. Uma autenticação é composta por 5 mensagens, de acordo com a especificação do OpenID 2.0, o que totaliza 10.000 mensagens por cliente. Devido a limitações de tempo e alocação de recursos, a média de autenticações por segundo foi calculada a partir de 5 execuções para cada configuração de cliente (e.g. 20). Apesar de representar um número relativamente baixo de execuções para cada configuração, durante os testes e experimentos foi observado um comportamento razoavelmente estável, sem variações significativas entre uma e outra execução.

O número de autenticações por segundo varia de 860 (20 clientes) a 995 (80 clientes) no ambiente UFAM-VMs. Por outro lado, com 100 clientes há uma queda

Tabela 5. Número de autenticações por segundo com 20, 40, 80 e 100 clientes

Ambiente	20 clientes	40 clientes	80 clientes	100 clientes
UFAM-VMs	867.73	984.59	995.12	960.11
Amazon-EC2	1969.17	2166.58	2244.30	2244.04
Amazon-DCs	26.66	50.72	92.42	114.05

no desempenho. Ela deve-se às limitações do ambiente, que entra num estado de *thrashing* (i.e. sobrecarga de escalonamento, acesso concorrente de I/O, etc.) com um número elevado de clientes. Um comportamento similar é observado no ambiente Amazon-EC2. Portanto, para executar mais clientes seriam necessárias mais VMs, como foi analisado e constatado na prática.

É interessante observar que ambos os ambientes (UFAM-VMs e Amazon-EC2) são similares, exceto pelo fato das VMs do Amazon-EC2 terem um poder de computação superior às do UFAM-VMs. Portanto, essa disparidade justifica a diferença de desempenho. A exemplo, se pegarmos o caso de 80 clientes, o Amazon-EC2 suporta uma carga de autenticações/s 2.25 vezes maior que o UFAM-VMs.

Diferentemente dos dois primeiros ambientes, o Amazon-DCs apresenta um desempenho significativamente menor. A principal razão para esse desempenho está na latência da rede (WAN) entre os datacenters, como discutido a seguir. Com 20 clientes chega-se a uma vazão de 26.66 autenticações/s, enquanto que com 100 clientes é possível atingir-se uma média de 114.05 autenticações/s.

Latência do ambiente e sistema. A latência da rede foi medida utilizando o comando `ping` para gerar 100 pacotes de 512 bytes. Portanto, a latência representa a média de 100 mensagens ICMP. O tamanho de 512 bytes deve-se ao fato de a maioria das mensagens de autenticação do OpenID girar em torno desse tamanho.

A latência entre datacenters, no ambiente Amazon DCs, varia significativamente. Ela vai de 32.10ms entre Oregon e N. California até 87.34ms entre Oregon and N. Virginia, o que representa uma diferença de 2.72 vezes. Portanto, uma das coisas a observar, na escolha dos datacenters, é a latência da rede. Inter-conexões privadas e/ou de menor latência irão gerar ganhos significativos no desempenho geral do sistema. Por outro lado, as latências de rede dos ambientes UFAM-VMs e Amazon-EC2 ficaram em uma média de aproximadamente 0.075ms e 0.185ms, respectivamente. Como observado, a latência entre os datacenters é superior às demais, o que justifica os números de vazão do sistema no ambiente multi-datacenter.

Adicionalmente, a média de latência das autenticações (média de 9.000) ficou em 7.16ms (UFAM-VMs), 6.22ms (Amazon-EC2) e 857.93ms (Amazon-DCs). Como esperado, a latência de autenticação no ambiente Amazon-DCs é significativamente alta. Comparando os ambientes UFAM-VMs e Amazon-EC2, novamente tem-se um desempenho melhor do segundo em relação ao primeiro. Entretanto, observa-se que a diferença é muito menor quando equiparada a diferença de vazão do sistema, o que significa que o maior impacto na vazão do sistema está mais relacionado com a capacidade de computação do que com latência de uma autenticação.

Capacidade e dimensionamento do sistema. Coletando os dados de um ambiente universitário real, com aproximadamente 11.5k usuários, pôde ser constatado que: (1) o número máximo de autenticações, na hora mais crítica num período de 7

dias, chega a 143907 autenticações, o que representa uma média de 39.97 autenticações por segundo; e (2) o maior pico de autenticações, somando o pior segundo dos dois sistemas de autenticação, o Active Directory e o LDAP, chega a 118 autenticações por segundo. Entretanto, mais de 99.99% do tempo o número de autenticações por segundo fica abaixo de 100. Cabe ressaltar ainda que esses números representam as demandas de autenticação de praticamente todos os sistemas da instituição, incluindo login em computadores de laboratórios ou de uso pessoal, logins de acesso à rede (e.g. autenticações 802.1x), autenticações em sistemas Web online, verificações de usuários nos servidores SMTP, entre outros.

A partir dos números apresentados, pode ser inferido que a solução desenvolvida é capaz de suportar a demanda de ambientes com algumas dezenas de milhares de usuários. Considerando o pior caso, de 118 autenticações por segundo, o ambiente UFAM-VMs comporta aproximadamente 97k usuários. Já o ambiente Amazon-EC2 comporta uma instituição, ou sistema, com mais de 218k usuários. Por outro lado, o ambiente Amazon-DCs comporta uma infraestrutura com apenas cerca de 11k usuários. Mesmo assim, considerando a robustez provida pelo ambiente, os números são interessantes, mostrando que a solução é aplicável a ambientes reais. Cabe ressaltar que esses números representam uma única instância do sistema. Com múltiplas instâncias, e distribuição de carga entre elas, a capacidade do sistema deve potencialmente aumentar de forma linear.

6. Conclusão

O artigo detalhou uma arquitetura para o desenvolvimento de provedores de identidade mais resilientes e seguros. Entre os componentes e técnicas fundamentais estão protocolos para tolerar falhas arbitrárias, diversidade de sistemas operacionais e *hypervisors*, componentes seguros para garantir a confidencialidade de dados sensitivos e um novo elemento, denominado de gateway, para manter compatibilidade com infra-estruturas OpenID existentes.

Um protótipo foi implementado, como prova de conceito, seguindo a especificação do OpenID 2.0. A base de desenvolvimento foi a linguagem Java e as bibliotecas *openid4java*, BFT-SMaRt e BouncyCastle.

No que refere-se aos resultados, são discutidos os mecanismos e recursos adotados para evitar e/ou mitigar diferentes tipos de ataques, bem como tolerar falhas arbitrárias no serviço OpenID. Adicionalmente, a avaliação de desempenho do sistema foi baseada em três ambientes distintos, sendo um deles multi-data center. Os resultados de vazão e latência demonstram que o sistema é capaz de atender demandas de ambientes reais, como instituições de ensino, com mais de 200k usuários.

Agradecimentos

Agradecemos aos revisores anônimos pelos comentários. Este trabalho é suportado pela FCT, através do Programa Multianual (LaSIGE) e projeto TRONE (CMU-PT/RNQ/0015/2009), pela Comissão Européia, através do projeto SecFuNet (FP7-ICT-STREP-288349), e pelo CNPq, através dos processos de número 590047/2011-6 e 202104/2012-5.

Referências

Amazon Web Services, Inc. (2014). Amazon EC2 Pricing. `http://aws.amazon.com/ec2/pricing/`.

Barreto, L., Siqueira, F., da Silva Fraga, J., and Feitosa, E. (2013). Gerenciamento de Identidades Tolerante a Intrusões. In *Anais do XXXI SBRC*. SBC.

Bessani, A., ao Sousa, J., and Alchieri, E. (2013). State Machine Replication for the Masses with BFT-SMaRt. Technical report, DI/FCUL. `goo.gl/XPEHMm`.

de Sousa, J. C., Bessani, A., and Sousa, P. (2010). Typhon: um serviço de autenticação e autorização tolerante a intrusões. In *INForum*. `http://goo.gl/mgPT7Y`.

Kreutz, D., Feitosa, E., Malichevskyy, O., Barbosa, K. R. S., and Cunha, H. (2013a). Um modelo funcional para serviços de identificação e autenticação tolerantes a faltas e intrusões. In *SBSeg*. SBC. `goo.gl/yDet4k`.

Kreutz, D., Malichevskyy, O., Feitosa, E., Barbosa, K. R. S., and Cunha, H. (2014). System design artifacts for resilient identification and authentication infrastructures. In *Proceedings of the Tenth International Conference on Networking and Services*. To appear.

Kreutz, D., Niedermayer, H., Feitosa, E., da Silva Fraga, J., and Malichevskyy, O. (2013b). Architecture components for resilient networks. Technical report, SecFuNet Consortium. `goo.gl/xBHCNb`.

Kreutz, D., Ramos, F. M., and Verissimo, P. (2013c). Towards secure and dependable software-defined networks. In *SIGCOMM HotSDN*, pages 55–60.

Malichevskyy, O., Kreutz, D., Pasin, M., and Bessani, A. (2012). O vigia dos vigias: um serviço RADIUS resiliente. In *INForum*. `http://goo.gl/GDUYBw`.

Niedermayer, H., Kreutz, D., Feitosa, E., Malichevskyy, O., Bessani, A., Fraga, J., Cunha, H. A., and Kinkelin, H. (2014). Trustworthy and resilient authentication service architectures. Technical report, SecFuNet Consortium.

of the Bouncy Castle Inc., L. (2014). The Legion of the Bouncy Castle. `https://www.bouncycastle.org`.

openid-server Community (2010). JOIDS. `goo.gl/nkyZIj`.

OpenID4Java (2013). OpenID 2.0 Java libraries. `goo.gl/c3kFV`.

Prince, M. (2012). Ceasefires Don't End Cyberwars. `goo.gl/GI506`.

Prince, M. (2013). The DDoS that almost broke the internet. `goo.gl/g5Qs1`.

Recordon, D. and Reed, D. (2006). Openid 2.0: A platform for user-centric identity management. In *ACM WDIM*, DIM '06, pages 11–16. ACM.

Rescorla, E. (1999). Diffie-Hellman Key Agreement Method. `goo.gl/Ta2jhI`.

Urien, P. and Dandjinou, M. (2006). Introducing smartcard enabled radius server. In *International Symposium on CTS*, pages 74–80.

Uruena, M., Munoz, A., and Larrabeiti, D. (2012). Analysis of privacy vulnerabilities in single sign-on mechanisms for multimedia websites. *Multimedia Tools and Applications*.

Verissimo, P. E., Neves, N. F., Cachin, C., Poritz, J., Powell, D., Deswarte, Y., Stroud, R., and Welch, I. (2006). Intrusion-tolerant middleware: The road to automatic security. *Security & Privacy, IEEE*, 4(4):54–62.

Viejo, A. (2014). Microsemi Enables FPGA-Based Root-of-Trust Solution for Embedded Systems with Introduction of Secure Boot Reference Design. `goo.gl/OYCDNT`.

Índice por Autor

A

Alchieri, E. 102
Almeida, V. 133
Arantes, L. 74

B

Baggio, M.A. 88
Barros, F. 3
Basso, T. 45
Baude, F. 61
Bernardino, M. 17
Braga, M. 102

C

Cechin, S.L. 147
Cunha, H. 174

D

De Bona, L.C. 31
de Vit, A.R.D. 88
Dobler, R. 147
Duarte Jr., E. 31, 74
Duarte, P.A. 133

F

Feitosa, E. 174

G

Granville, L.Z. 133
Greve, F. 116
Guarienti, P. 17

K

Khouri, C. 116
Koslovski, G. 160
Kreutz, D. 174

M

Macedo, D.F. 133
Mello, V. 45
Moraes, R. 45
Mota, V. 133

N

Netto, J. 147
Nobre, J. 133
Nogueira, J.M. 133
Nunes, R.C. 88

O

Obelheiro, R. 160
Oliveira, F.M 17

P

Pasin, M. 88
Prudêncio, R. 3

R

Rodrigues, L. 74
Ruoso, V.K. 31

S

Salvador, E. 133
Silva, M. 160
Souza, L. 3
Souza, M. 61

V

Vidal, W. 147

W

Weber, T. 147

Z

Zorzo, A. 17

Promoção

SBC Sociedade Brasileira de Computação

LARC

Organização

UFSC

DAS

ine CTC • UFSC

Patrocínio

cgi.br Comitê Gestor da Internet no Brasil

nic.br Núcleo de Informação e Coordenação do Ponto BR

CAPES

Microsoft Research

FAPESC

CNPq Conselho Nacional de Desenvolvimento Científico e Tecnológico

Finep INOVAÇÃO E PESQUISA

Apoio

BRDigital Telecomunicações

RNP

ISSN 2177-496X

9 772177 496009 >