31º Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos

6 a 10 de Maio de 2013
Brasília-DF

Anais
Workshop de Testes e Tolerância a Falhas

Editora
Sociedade Brasileira de Computação (SBC)

Organizadores
Eduardo Adilio Pelinson Alchieri (UnB)
Francisco Vilar Brasileiro (UFCG)
Jacir Luiz Bordim (UnB)
Rafael Timóteo de Sousa Júnior (UnB)
William Ferreira Giozza (UNB)

Realização
Universidade de Brasília (UnB)

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

Capa: João Paulo Andrade Lima

Produção Editorial: Eduardo Adilio Pelinson Alchieri
Marcos Fagundes Caetano

Workshop de Testes e Tolerância a Falhas (14 : 2013: Brasília, DF)

Anais / Workshop de Testes e Tolerância a Falhas; organizado por Jacir Luiz Bordim... [et al.] — Porto Alegre: SBC, c2013

105 p.

WTF 2013
Realização: Universidade de Brasília
ISSN: 2177-496X

1. Redes de Computadores - Congressos. 2. Sistemas Distribuídos - Congressos. I. Bordim, Jacir Luiz. 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)

Diretor Administrativo
Luciano Paschoal Gaspary (UFRGS)

Diretor de Finanças
Luci Pirmez (UFRJ)

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

Diretora de Educação
Mirella Moura Moro (UFMG)

Diretora de Publicações
Karin Koogan Breitman (PUC-Rio)

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

Diretora de Secretarias Regionais
Thais Vasconcelos Batista (UFRN)

Diretor de Divulgação e Marketing
Edson Norberto Cáceres (UFMS)

Diretor de Relações Profissionais
Roberto da Silva Bigonha (UFMG)

Diretor de Competições Científicas
Ricardo de Oliveira Anido (UNICAMP)

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

Promoção

Conselho

Mandato 2011-2015

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

Mandato 2009-2013

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

Suplentes – 2011-2013

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

Laboratório Nacional de Redes de Computadores (LARC)

Diretoria

Diretor do Conselho Técnico-Científico

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 e UFS.

Realização

Comitê de Organização

Coordenação Geral

Jacir Luiz Bordim (UnB)

Rafael Timóteo de Sousa Jr (UnB)

William F. Giozza (UnB)

Coordenação do Comitê de Programa

Carlos André Guimarães Ferraz (UFPE)

José Augusto Suruagy Monteiro (UFPE)

Coordenação de Palestras e Tutoriais

Nelson Luis Saldanha da Fonseca (UNICAMP)

Coordenação de Painéis e Debates

Lisandro Zambenedetti Granville (UFRGS)

Realização

Coordenação de Minicursos

Joni da Silva Fraga (UFSC)

Coordenação de Workshops

Francisco Vilar Brasileiro (UFCG)

Coordenação do Salão de Ferramentas

Gustavo Sousa Pavani (UFABC)

Comitê Consultivo

Artur Ziviani (LNCC)
Bruno Schulze (LNCC)
Célio Vinícius Neves de Albuquerque (UFF)
Dorgival Guedes (UFMG)
Elias Procópio Duarte Jr (UFPR)
José Ferreira de Rezende (UFRJ)
Jussara Almeida (UFMG)
Ronaldo Alves Ferreira (UFMS)

Realização

Organização Local

Aletéia Patrícia Favacho de Araújo (UnB)
André Costa Drummond (UnB)
Divanilson Rodrigo de Sousa Campelo (UnB/UFPE)
Edna Dias Canedo (UnB)
Eduardo Adilio Pelinson Alchieri (UnB)
Flávio Elias Gomes de Deus (UnB)
Jacir Luiz Bordim (UnB)
Marcos Fagundes Caetano (UnB)
Maristela Terto de Holanda (UnB)
Priscila América Solís Mendez Barreto (UnB)
Rafael Timóteo de Sousa Junior (UnB)
William Ferreira Giozza (UnB)

Mensagem dos Coordenadores Gerais

Sejam bem vindos ao **31º Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos** (**SBRC 2013**), realizado pela primeira vez na Capital Federal, Brasília, DF.

Manter o SBRC na posição do mais importante evento científico nacional em Redes de Computadores e Sistemas Distribuídos e um dos maiores da área de Informática no país é um desafio importante assumido por nós da Universidade de Brasília. Além de estimular a troca de ideias e experiências, a discussão de temas de pesquisa avançados, como é sua tradição, o **SBRC**, em sua **trigésima primeira edição**, está organizado para proporcionar uma interação proveitosa entre estudantes, professores, pesquisadores e profissionais com interesses na área do evento.

O **SBRC 2013** está com uma programação bastante variada e com alta qualidade técnica. O número de **sessões técnicas** na trilha principal foi ampliado para 24 de modo a permitir a apresentação de **73 artigos completos**, cobrindo uma ampla lista de tópicos relevantes e atuais nas áreas de Redes de Computadores e Sistemas Distribuídos. O Salão de Ferramentas apresenta **11 ferramentas** que serão debatidas e demonstradas em um fórum específico. Este ano, os estudantes participantes do SBRC serão contemplados com a oferta de **7 minicursos** versando sobre temas atuais normalmente não abordados nas grades curriculares. A programação do SBRC 2013 inclui ainda **4 palestras convidadas** proferidas por pesquisadores de alto renome internacional e **3 painéis** abordando temas atuais e polêmicos. Além dessas atividades tradicionais, o SBRC 2013 abriga a realização de **9 workshops** que ocorrem em paralelo com o evento: XVIII Workshop de Gerência e Operação de Redes e Serviços (WGRS), XIV Workshop da Rede Nacional de Ensino e Pesquisa (WRNP), XIV Workshop de Testes e Tolerância a Falhas (WTF), XI Workshop de Computação em Grade e Aplicações (WCGA), IX Workshop de Redes P2P, Dinâmicas, Sociais e Orientadas a Conteúdo (WP2P+), IV Workshop de Pesquisa Experimental na Internet do Futuro (WPEIF), III *Workshop on Autonomic Distributed Systems* (WoSiDA), III Workshop de Redes de Acesso em Banda Larga (WRA) e o I ***Workshop of Communication in Critical Embedded Systems* (WoCCES)**. Por fim, o SBRC 2013 promove um **Fórum Governo** no intuito de reunir profissionais, acadêmicos e gestores com a finalidade de discutirem os principais desafios e as soluções avançadas de redes e sistemas distribuídos para as diversas esferas governamentais.

O **SBRC 2013** retoma a discussão sobre os caminhos da pesquisa e da inovação em redes de computadores e sistemas distribuídos no País. Em particular conta com um **painel** que visa discutir os **grandes desafios da área** com representantes da indústria e do governo. Além disso, para homenagear aqueles que contribuíram significativamente para o desenvolvimento da pesquisa e o fortalecimento da comunidade científica nas áreas de Redes de Computadores e Sistemas Distribuídos no Brasil, o SBRC 2013 mantém o **Prêmio Destaque SBRC**, nesta edição, homenageando a professora Liane Margarida Rockenbach Tarouco, da Universidade Federal do Rio Grande do Sul, em reconhecimento às suas contribuições científicas e aos serviços prestados em benefício do SBRC e de sua comunidade.

A organização de um evento do porte e da importância do SBRC só pode ser realizada se contar com a ajuda de uma equipe qualificada e dedicada.

Mensagem dos Coordenadores Gerais

Gostaríamos de agradecer aos membros dos **Comitês de Organização Geral e Local** pelo trabalho voluntário de excelente qualidade e pelo apoio incansável durante as várias etapas da organização deste evento. Somos muito gratos também ao apoio da **SBC** e do **LARC**, promotores do SBRC. Em particular agradecemos aos membros do **Conselho Consultivo do SBRC** e da coordenação da **Comissão Especial de Redes de Computadores e Sistemas Distribuídos** da SBC, pela confiança depositada e pelo suporte financeiro inicial indispensável para a realização do SBRC 2013. Gostaríamos de agradecer ainda ao **Comitê Gestor da Internet no Brasil**, às agências governamentais de fomento – **CNPq** e **CAPES**, e aos **patrocinadores** por reconhecerem e valorizarem a importância do SBRC como fórum de divulgação da pesquisa e inovação em redes de computadores e sistemas distribuídos. Por fim, nossos agradecimentos aos **Departamentos de Engenharia Elétrica** e de **Ciência da Computação** da **UnB**, por apoiarem nossa dedicação para viabilizar a realização deste evento.

Em nome do Comitê Organizador do SBRC 2013, desejamos a todos uma semana bastante produtiva e agradável.

Jacir Luiz Bordim
Rafael Timóteo de Sousa Junior
William Ferreira Giozza
Coordenadores Gerais do SBRC 2013

Mensagem do Coordenador do WTF 2013

Sejam bem vindos ao XIV Workshop de Testes e Tolerância a Falhas (WTF). O WTF é um evento que ocorre anualmente, sendo promovido pela Comissão Especial de Sistemas Tolerantes a Falhas (CE-TF) da Sociedade Brasileira de Computação (SBC). A sua décima quarta edição ocorre neste ano de 2013 em Brasília-DF, em conjunto com o SBRC (Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos).

O WTF constitui-se em um fórum de debates e apresentações de trabalhos de pesquisas relacionadas à confiança no funcionamento de sistemas, em especial nas áreas de testes e tolerância a falhas e intrusões. Nesta edição, o WTF contemplará atividades de apresentação de artigos, além de uma palestra convidada. A programação do XIV WTF inclui 7 trabalhos que foram selecionados para publicação e apresentação. Cada trabalho submetido ao WTF teve no mínimo 3 revisões, realizadas por um Comitê de Avaliação criterioso composto por 32 pesquisadores especialistas nas diversas áreas de interesse do workshop. Os artigos aceitos foram divididos em três sessões técnicas: Arquiteturas de Software (2 artigos); Algoritmos Distribuídos (3 artigos); e Escalonamento e Virtualização (2 artigos).

Em complemento as sessões técnicas, para enriquecer e fomentar ainda mais as discussões, convidamos a professora Luciana Arantes, pesquisadora do laboratório LIP6 (*Laboratoire d'Informatique de Paris 6*) vinculado à Universidade Paris 6 (*University of Pierre et Marie Currie*), para realizar uma palestra sobre MapReduce tolerante a falhas bizantinas. Esperamos que os trabalhos selecionados e a palestra convidada suscitem discussões e interações frutuosas entre os participantes do workshop.

Gostaria de agradecer a todo o Comitê de Programa e Revisores pelo excelente trabalho de avaliação realizado, pela reatividade e boas discussões, que contribuíram para termos uma seleção criteriosa e construtiva. Agradeço também os autores e a palestrante convidada pelas suas valiosas contribuições. Gostaria de agradecer a Comissão Organizadora do SBRC 2013 e ao Francisco Brasileiro, Coordenador de Workshops, por nos acolher e fornecer o apoio logístico para a concretização do evento. Finalmente, um agradecimento especial para Fabíola Greve, Coordenadora do WTF 2012, pela presteza e apoio constante na execução deste trabalho.

Eduardo Adilio PelinsonAlchieri
Coordenador do WTF 2013

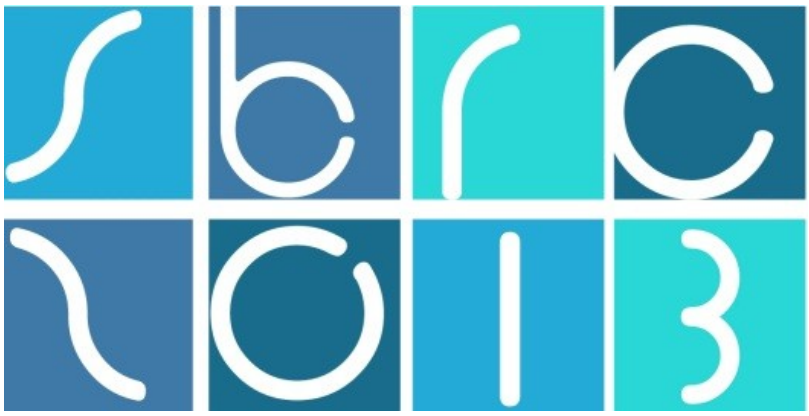
Comitê de Programa do WTF 2013

Alcides Calsavara (PUC-PR)
Alexandre Sztajnberg (UERJ)
Alysson Bessani (Universidade de Lisboa)
Ana Ambrosio (INPE)
Avelino Zorzo (PUC-RS)
Carlos Montez (UFSC)
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 (PUC-RS)
Frank Siqueira (UFSC)
Irineu Sotoma (UFMS)
Jose Pereira (Universidade do Minho)
Lasaro Camargos (UFU)
Lau Cheuk Lung (UFSC)
Lúcia Drummond (UFF)
Luciana Arantes (Universidade Paris 6)
Marco Vieira (universidade de Coimbra)
Michele Nogueira (UFPR)
Miguel Correia (Instituto Superior Técnico)
Patricia Machado (UFCG)
Raimundo Barreto (UFAM)
Raul Ceretta Nunes (UFSM)
Regina Moraes (UNICAMP)
Rogerio de Lemos (Universidade de Kent)
Rui Oliveira (Universidade do Minho)
Taisy Weber (UFRGS)
Udo Fritzke Jr. (PUC-MG)

Revisores do WTF 2013

Alcides Calsavara (PUC-PR)
Alexandre Sztajnberg (UERJ)
Alysson Bessani (Universidade de Lisboa)
Ana Ambrosio (INPE)
Antonio Augusto Ribeiro Coutinho (UEFS)
Carlos Montez (UFSC)
Daniel Batista (USP)
Daniel Cordeiro (USP)
Daniel Fernandes Macedo (UFMG)
Eduardo Alchieri (UnB)
Eduardo Bezerra (UFSC)
Eliane Martins (UNICAMP)
Elias P. Duarte Jr. (UFPR)
Fernando Dotti (PUC-RS)
Frank Siqueira (UFSC)
Irineu Sotoma (UFMS)
Jose Pereira (Universidade do Minho)
Lasaro Camargos (UFU)
Luciana Arantes (Universidade Paris 6)
Marcelo Ribeiro Xavier da Silva (UFSC)
Marco Vieira (universidade de Coimbra)
Michele Nogueira (UFPR)
Miguel Correia (Instituto Superior Técnico)
Patricia Machado (UFCG)
Raimundo Barreto (UFAM)
Raul Ceretta Nunes (UFSM)
Rogerio de Lemos (Universidade de Kent)
Udo Fritzke Jr. (PUC-MG)

Sumário



31º Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos
Brasília-DF

XIV Workshop de Testes e Tolerância a Falhas

Sessão Técnica 1

Arquiteturas de Software

Proposta para Atualização de SGBDs para Aplicações com Demanda de Contínua Disponibilidade usando Suporte do Modelo de Componentes de *Software*

Cleandro Flores De Gasperi, Marcia Pasin[1]

[1]Programa de Pós-Graduação em Informática (PPGI)
Universidade Federal de Santa Maria (UFSM)
Av. Roraima 1.000 – 97.105-900
Cidade Universitária – Santa Maria – RS – Brasil

cleandro@cpd.ufsm.br, marcia@inf.ufsm.br

Abstract. *Database management system (DBMS) is a critical part in enterprise software and its on-the-fly updating is not trivial due its inherent complexity and requirements such as continuous availability. Current solutions to enterprise software update are costly and involve human intervention. In this paper, we propose the use of component model as a support to achieve continuous availability in dynamic software update (DSU) in DBMS. The component model allows to encapsulate implementation details, which is an interesting property to software replacement. This solution provides as benefits the absence of additional hardware and reduced service unavailability. To validate our proposal, a prototype was developed using Fractal component model. The experimental evaluation confirms the effectiveness of our approach but, as expected, indicates that the component model itself is still a bottleneck.*

Resumo. *Sistema de gerenciamento de banco de dados (SGBD) é parte fundamental do software corporativo e sua atualização dinâmica não é simples devido à sua complexidade. Frequentemente, atualização de SGBD é manualmente executada e está associada à indisponibilidade de serviço e uso de hardware adicional. Este artigo propõe o uso de modelo de componentes de software como alternativa para a aplicação de atualização dinâmica de software (ADS) em um SGBD. O modelo de componentes apresenta um nível de abstração que favorece a troca de componentes pois oculta detalhes de implementação. ADS apresenta como benefícios a ausência da necessidade de hardware adicional e da indisponibilização do serviço. Para validação da proposta foi desenvolvido um protótipo utilizando o modelo de componentes Fractal. Testes em ambiente controlado confirmam a viabilidade da solução mas indicam que o próprio modelo de componentes ainda é um gargalo.*

1. Introdução

A tarefa de atualização de *software* é necessária porque apesar do contínuo esforço no desenvolvimento e uso de novas ferramentas e técnicas de programação, o código está sempre sujeito ao envelhecimento e à existência de *bugs*. Para sobreviver, um *software* está em continuo desenvolvimento. Novas distribuições e versões são sistematicamente

disponibilizadas para corrigir eventuais *bugs* e implementar necessidades não cobertas pela implementação anterior.

Contudo, a experiência mostra que o processo de atualização do *software* pode sofrer problemas como falha durante a atualização (causando indisponibilidade parcial ou total do sistema), indisponibilização de serviço durante a atualização (ainda que em um cenário livre de falhas), inserção de novos *bugs* após atualização, etc. Na prática, esses problemas são tratados de forma restrita ou ignorados. O ideal seria que sistemas computacionais permitissem mecanismos automáticos, com a implementação de serviços para garantia de um conjunto amplo de propriedades específicas. Neste contexto, Atualização Dinâmica de *Software* (ADS) aparece como uma resposta.

ADS [Stoyle et al. 2007] é o mecanismo que possibilita substituição automática de programas mantendo a disponibilidade do serviço. Existem soluções para ADS que contemplam diferentes áreas da computação como sistemas operacionais (atualização automática do *Windows*, *Mac OS X*, distribuições de Linux), linguagens de programação (substituição de funções em linguagem C [Segal e Frieder 1993]) e engenharia de *software* (notadamente soluções com troca de componentes [Leger et al. 2007, Wang et al. 2006]). Entretanto, muitas soluções adotadas na prática, como aquelas aplicadas a sistemas operacionais, ainda necessitam interferência humana ou reinicializarão do serviço para disponibilizar novas atualizações.

O conjunto de propriedades que sistemas com ADS deveriam garantir foram apresentadas anteriormente em [Segal e Frieder 1993]. Estas propriedades contemplam (i) transparência para ocultar ao usuário a ocorrência de uma atualização do sistema, (ii) automação de forma a minimizar intervenção humana evitando erros e tornando o processo de atualização mais confortável para o usuário, (iii) suporte para reestruturação de código possibilitando a inclusão e remoção de módulos, (iv) descarte da necessidade de *hardware* adicional evitando maximização de custos, (v) ausência de restrição de linguagem e ambiente, flexibilizando portabilidade para diferentes ambientes operacionais e linguagens, (vi) facilidade de manutenção, pois a evolução de *software* ocorre de forma contínua, e (vii) disponibilidade contínua pela troca de componentes ocorre sem interrupção total de serviço. A implementação dessas propriedades não é tarefa trivial e envolve diferentes aspectos em um projeto complexo. Por isso, sistemas com ADS focam em um conjunto restrito de propriedades.

Outro grande desafio, no contexto de ADS, é a ausência de soluções genéricas e automatizadas para ambientes mais amplos e complexos, como sistemas corporativos e $clouds$. Apesar de ADS não ser uma novidade (a pesquisa mais remota data da década de 1970 [Fabry 1976]), seu uso em aplicações empresariais é limitado. Servidores de aplicação e sistemas de gerenciamento de banco de dados (SGBD) integram parte fundamental do *software* de sistemas corporativos e empresariais e sua atualização não é trivial. Sistemas empresariais requerem soluções para tratar escalabilidade, elasticidade, dinamismo, heterogeneidade e, portanto, complexidade. Tipicamente, aplicações são implementadas em múltiplos *tiers* onde cada *tier* fornece um serviço específico. Tratar tais adversidades sem comprometer a disponiblilidade e desempenho é tarefa complicada. Nesses ambientes, manutenção e atualização são tarefas manuais ou parcialmente automatizadas.

Frequentemente soluções para atualização de *software* adotadas por instituições de

grande porte estão focadas no uso de *hardware* adicional (o serviço executa em um conjunto de máquinas auxiliares enquanto o sistema principal é atualizado) e na interferência humana. Esta estratégia impacta sensivelmente no custo do processo de substituição de *software* e envolve amplo planejamento de uma equipe para que o serviço não seja interrompido. Instituições de médio e pequeno porte tipicamente optam por mecanismos simples (parada total do serviço) mas comprometem a disponibilidade das aplicações. A solução mais adequada seria aplicada de forma automática, com baixo custo de execução e com possibilidade de realizar atualização de *software* de forma gradativa, sem interromper o serviço para os clientes e sem uso de *hardware* adicional. Adicionalmente, a reduzida interferêcia humana evitaria a inclusão de erros durante esse processo.

Neste sentido, este trabalho apresenta um passo à frente na atualização de *software* para ambiente empresarial e propõe o uso do modelo de componentes de *software* como suporte para aplicação de ADS em SGBD. Esta proposta é inédita. Um SGBD é um *software* complexo que possui muitos módulos executando tarefas importantes: otimização de consultas, gerenciamento transacional, provimento de persistência, controle de acesso concorrente, entre outros. No contexto de aplicações e sistemas empresariais, o modelo de componentes de *software* aparece como alternativa para tratar a complexidade inerente, mas, em contra-partida, adiciona o *overhead* da implementação da própria abstração de componentes. Um componente de *software* é uma unidade independente e encapsulada que possibilita ocultar detalhes de implementação. Destaca-se que, de fato, esta solução é interessante porque a ADS se beneficia amplamente do modelo de componentes de *software*. Como o sistema é representado através de um conjunto de componentes independentes, um componente pode ser substituído enquanto os demais continuam a operar, ainda que a disponibilidade do serviço como um todo seja reduzida e sem a necessidade de *hardware* adicional. Outro benefício é que esta solução é genérica e pode ser reaproveitada em diversas instalações.

Apesar da limitação tecnológica (não existe atualmente implementação de SGBD no modelo de componentes), o fato do SGBD ser modelado como um conjunto de componentes possibilitaria substituição gradual de *software* sem interrupção total do serviço. O modelo de componentes permite que seja associado ao sistema-alvo um mecanismo de descrição de arquitetura [Bass et al. 2003] favorecendo a análise das associações entre módulos do SGBD. Uma discussão sobre os benefícios do uso do modelo de componentes e da descrição de arquitetura para atualização de *software* é apresentada em [Oreizy et al. 1999]. Neste trabalho, o modelo de componentes Fractal [Bruneton et al. 2006] é usado para implementar a descrição da arquitetura.

Mais especificamente, este trabalho objetiva: (i) propor um modelo genérico para representar um SGBD visando a arquitetura de *software* baseada no modelo de componentes de Fractal que permita a substituição destes, (ii) apresentar uma lógica para a substituição gradativa de componentes sem parada total do serviço para os clientes, sem a necessidade de *hardware* adicional, e que não onere o desempenho esperado demasiadamente, (iii) codificar um esboço de um protótipo para mostrar a viabilidade desta solução.

O restante do artigo está organizado como segue. Seção 2 discute trabalhos relacionados. Seção 3 apresenta a proposta de ADS para SGBD com suporte do modelo de componentes. Implementação de protótipo e o modelo de componentes Fractal são

descritos na seção 4. A avaliação experimental, conduzida através do protótipo implementado, e descrição dos resultados são apresentadas na seção 5. A seção 6 conclui o artigo apresentando considerações finais e perspectivas para trabalhos futuros.

2. Trabalhos Relacionados

De acordo com [Wahler et al. 2009], existem dois principais tipos de soluções para ADS: soluções baseadas em rotinas e soluções baseadas em componentes. Ainda pode ser acrescentado um terceiro tipo que agrega soluções baseadas em arquitetura de *software*. Este último tipo permite não apenas o controle dos componentes mas também das vinculações entre os componentes, o que representa um suporte interessante para implementação de ADS.

Sistemas para atualização de *software* baseados em rotinas são atrelados a detalhes de implementação, o que limita a portabilidade dessas soluções. Um exemplo de solução nesta linha é PODUS. PODUS [Segal e Frieder 1993] é uma ferramenta capaz de promover ADS na camada do sistema operacional. Uma rotina em execução somente é atualizada quando estiver inativa. A atualização inicia carregando em outro espaço de memória novas versões das rotinas a serem atualizadas. A seguir o programa é interrompido e a sua pilha de execução é analisada. Então, redirecionamentos são executados através de uma tabela de indireção. Na próxima vez que a rotina for chamada, será direcionada para a nova versão descrita na tabela. Quando todas as rotinas previstas na atualização estiverem na nova versão, o programa é considerado atualizado. [Lyu et al. 2001] usa solução análoga mas difere porque a modificação de rotina ocorre de forma direta, sem precisar de uma tabela de indireções. Em outras palavras, dentro do código da versão antiga da rotina estará o endereçamento para a nova versão. ADS é realizada por chamadas diretas do sistema operacional, sem necessidade de *software* adicional. Finalmente, PROTEUS [Stoyle et al. 2007] oportuniza ADS em linguagens *C-like*, e atualização pode ser aplicada para tipos de dados e funções. A nova versão é escrita com PROTEUS, onde são explicitados tipos e funções a serem substituídas através de sintaxe específica. As atualizações são realizadas em tempo de execução.

Contrastando com soluções baseadas em rotinas, soluções baseadas em componentes ocultam detalhes de implementação e são mais genéricas. Um exemplo de serviço para ADS baseado no modelo de componentes e servidores de aplicação é [Wang et al. 2006], focando em objetos (mas especificamente Enterprise JavaBeans) que executam em servidores de aplicações. Outra proposta interessante [Leger et al. 2007] descreve a substituição de componentes genéricos aplicando as propriedades transacionais ACID para garantir um conjunto de propriedades requeridas pela ADS (reportadas neste artigo no item anterior). A ideia é substituir componentes dentro de um contexto transacional. Atualizações não executadas com sucesso podem ser revertidas com o auxílio de informações armazenadas em um *log*.

Existem soluções para ADS que adotam um modelo de maior granularidade que os apresentados anteriormente, com foco na descrição de arquitetura do sistema [Oreizy et al. 1998]. O principal diferencial de soluções arquiteturais está na possibilidade de reutilização da solução em outras implementações, desde que o *software* esteja de acordo com a arquitetura. O mesmo se aplica a soluções que seguem o modelo de componentes. Uma vantagem do modelo com descrição de arquitetura, em relação ao modelo

de componentes, é o suporte para o controle das dependências entre os componentes. Este suporte, que facilita o mapeamento de dependências, é crucial para a atualização de *software* por três motivos: (i) se um *software* depende de outro, a substituição deste código pode implicar na substituição de outro código, (ii) pelo suporte para tratamento de dependências externas (substituição de um *software* sendo correntemente usado por outro componente), e por possibilitar a substituição de um componente (primitivo) por dois ou mais componentes (componente composto) e vice-versa.

Todas as soluções existentes, sejam elas baseadas em rotinas, no modelo de componentes ou na descrição da arquitetura, implementam um conjunto restrito das propriedades requeridas para atualização de *software*. Em linhas gerais, soluções atreladas a rotinas e detalhes de implementação são mais eficientes em questão de desempenho mas sua portabilidade para outras plataformas é limitada. Em contraste, soluções baseadas em componentes de *software* e descrição de arquitetura são mais genéricas, mais flexíveis e com maior possibilidade de reaproveitamento em outros contextos.

Linguagens de programação também apresentam características positivas para a implementação de ADS. Em Java existem dois mecanismos importantes: (i) carga da classe (a partir do arquivo *.class*) e (ii) reflexão computacional. Como a carga de classe pode ser feita dinamicamente, a atualização de *software* pode ser vista como o carregamento de uma nova versão da classe antiga. Reflexão computacional é uma técnica de programação onde um sistema pode atuar sobre sua própria computação e se adaptar em presença de condições de mudança. A substituição de um componente antigo por outro componente é um tipo de adaptação. Entretanto, mecanismos importantes como tratamento de dependências entre componentes e transferência de estado ainda precisam ser tratados de forma explícita pelo programador. Finalmente, na literatura ainda existem outras soluções interessantes para atualização de *software*, mas todas elas tem restrições. Por exemplo, a plataforma OSGi permite suporte para substituição de módulos Java porém, como acontece em Java puro, o programador precisa mapear as dependências entre os módulos.

3. Proposta para Atualização de SGBD com o Modelo de Componentes

Esta seção apresenta a proposta genérica para a substituição de componentes sem parada total do SGBD e sem uso de *hardware* adicional. O elemento-chave desta proposta é o Gerenciador de Atualizações (GA) que executa a substituição de módulos do SGBD. A seguir, primeiramente são apresentadas limitações que possibilitaram estabelecer esta proposta de trabalho na tecnologia atual e em espaço tão curto de tempo e com reduzida equipe de desenvolvimento, e depois, de forma mais específica, é descrita a arquitetura do GA.

3.1. Limitações da proposta na tecnologia atual

Conforme comentado anteriormente, para que componentes do SGBD possam ser substituídos com o suporte do modelo de componentes Fractal ou outro modelo de componentes, é necessária a construção de um SGBD de acordo com o modelo de componentes de *software*, um tipo de construção atualmente inexistente. Embora existam implementações construídas de acordo com o paradigma de orientação a objetos (um exemplo é o HyperSQL), SGBDs atuais, em sua maioria, são construídos seguindo o modelo orientado

a processos. Estas implementações, se conjugadas com *frameworks* para construção de sistemas com o conceito de componentes de *software*, permitem obter uma solução compatível à proposta deste trabalho. Para tanto, possivelmente, será necessária a reescrita do SGBD (trabalho para uma grande equipe de desenvolvimento), de tal forma que os objetos que implementam seus módulos serão encapsulados em componentes.

Entretanto, para avaliar a proposta aqui apresentada, esta limitação foi contornada através do mapeamento dos módulos tradicionais que compõem um SGBD para componentes de *software*. A estrutura básica de um SGBD é descrita na literatura de sistemas de bancos de dados (veja [Elmasri e Navathe 2005], [Ramakrishnam e Gehrke 2003]). Aplicar a abstração de componentes de *software* sobre a estrutura do SGBD, encapsulando cada módulo em um componente é um caminho natural e, muito provavelmente, implementações futuras de SGBDs comerciais sigam o modelo de componentes, desde que esta abstração está se tornando muito popular na indústria de *software*. Assim, uma vez que módulos estarão encapsulados em componentes, operações de controle definidas (por Fractal, por exemplo) podem ser usadas para substituir dinamicamente componentes de *software*.

Outras limitações desta proposta são (i) execução de atualização de SGBD em um único nó (apenas um SGBD central é atualizado) e (ii) obrigatoriedade de manutenção da interface dos componentes. Entretanto, a solução aqui apresentada pode ser ampliada para suportar atualização em um ambiente mais complexo, com múltiplas instâncias de SGBDs executando simultaneamente. Finalmente, a obrigatoriedade de manutenção da interface dos componentes não invalida esta proposta e também foi utilizada em trabalho anterior [Wang et al. 2006]. De fato, muitas atualizações são feitas para consertar pequenos *bugs* (preservando a interface de serviços), não impactando significativamente no serviço oferecido pelo módulo ou na estrutura interna do sistema.

3.2. Arquitetura e Serviços

O gerenciador de atualizações (GA) possibilita que a atualização de componentes do SGBD seja realizada de forma gradativa e automática. Uma visão simplificada da arquitetura do GA é mostrada na Figura 1.

[Oreizy et al. 1999] destaca uma série de serviços importantes na atualização de *software*, que aqui são implementados pelo GA. Mais especificamente, o GA controla a versão dos componentes, identifica o momento adequado para realizar substituições, garante integridade estrutural do sistema, controla a demanda das requisições dos clientes e realiza a substituição de componentes propriamente dita. Esses serviços, executados por diferentes módulos, são brevemente abordados na sequência deste texto.

3.2.1. Serviço Gerenciador de Componentes

O controle de versão é uma tarefa fundamental para o funcionamento adequado da ADS e organiza as mudanças das informações dos componentes. No modelo aqui proposto, o GA utiliza um *log* para registrar componentes (módulos do SGBD) que foram substituídos, que representam uma variação particionada de um repositório. Ou seja, o próprio SGBD representa a última revisão do repositório. Em conjunto com o *log*, o GA usa uma tabela de versionamento. O GA identifica unicamente cada componente para o adequado

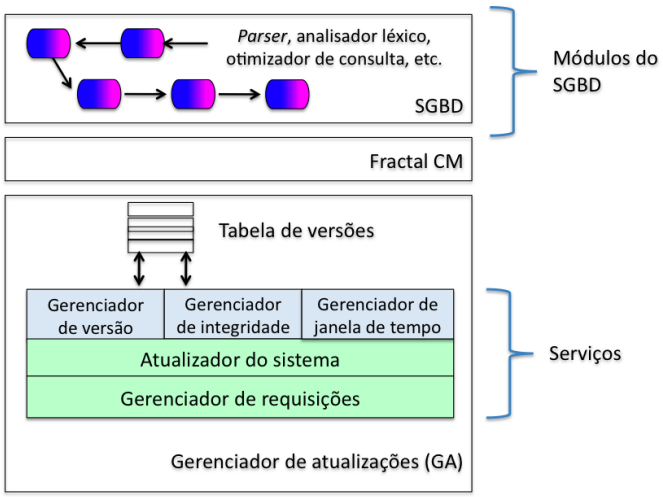


Figura 1. Esboço da arquitetura do GA

controle das versões, e armazena na tabela de versões informações sobre o a identificação do componente, o tipo de componente, e a versão que obedece uma ordem cronológica de finalização do componente, ou seja, o instante que o componente é liberado para a atualização no sistema. Internamente, o GA também mantém informação sobre as respectivas interfaces de cada componente. A tabela de versões tem três aplicações principais: (i) serve como base para recuperação do sistema, em caso de queda, (ii) facilita o processo de atualização dos componentes, pois mantém a versão atual de cada componente, e (iii) suporta um esquema de validação da interface implementada pelo novo componente.

3.2.2. Serviço Gerenciador de Janela de Tempo

O processo de atualização de componentes não deve onerar demasiadamente a atividade normal do sistema. Oportunidades para perceber o momento adequado para a realização desta tarefa devem ser continuamente buscadas. Se o serviço para os clientes estiver sofrendo grande demanda, a sobrecarga gerada pela atualização, mesmo que pequena, pode ser o acréscimo suficiente para comprometer a qualidade do serviço. Visando evitar sobrecarga, o GA identifica uma janela de tempo adequada para realizar a substituição de um componente. Informações obtidas por sensores, como taxa de ocupação da CPU, número total de transações ativas e estimativa de custo de transações podem ser usadas para mensurar a carga do sistema. Comparando estas medidas ou uma combinação delas com um parâmetro previamente definido e acessível ao GA, poderá ser realizada a escolha de uma janela de tempo adequada para efetuar o processo de atualização.

3.2.3. Serviço Gerenciador de Integridade

Componentes de *software* possuem interfaces bem definidas. A interface é uma coleção de operações que define os serviços disponibilizados. Assim, para garantir a integridade estrutural do sistema, é suficiente e necessário que o novo componente tenha a mesma interface do componente que está sendo substituído. É importante, salientar que é uma

validade de integridade estrutural e não comportamental do componente. O novo componente oferece as mesmas operações do antigo mas não existem garantias quanto ao seu funcionamento, ou seja, quanto aos resultados que serão produzidos por suas operações. Na proposta atual, a manutenção da integridade do sistema é provida através da implementação de uma tabela de versões dos componentes. Nesta tabela, o GA identifica quais interfaces o componente deve implementar. Então, pode ser assegurado que o novo componente implementa uma interface idêntica ou ao menos compatível com o antigo componente.

3.2.4. Serviço Gerenciador de Requisições de Clientes

Para [Wang et al. 2006] o cumprimento da premissa de que garante disponibilidade contínua significa que nenhuma requisição de cliente pode ser recusada durante o processo de ADS. Dessa forma, o GA intercepta e controla requisições encaminhadas ao sistema para assegurar a não interrupção de serviço para os clientes. Ao iniciar o processo de atualização de um componente, o GA espera requisições ativas finalizarem e enfileira as novas requisições, em um procedimento similar a [Wang et al. 2006]. Então, a ADS é executada. Após finalizar a ADS, o GA repassa a fila de requisições bloqueadas ao componente apropriado do SGBD.

4. Implementação

Para comprovação da viabilidade técnica desta proposta e serviços previamente descritos foi desenvolvido um esboço de um protótipo construído em linguagem *Java* 1.6 e a implementação de referência de Fractal *Julia* 2.5.1. O ambiente de desenvolvimento e compilação adotado foi o *IntelliJIDEA* 10.0.3 *Community Edition* da *JetBrains*. Uma breve descrição do modelo de componentes Fractal é apresentada no item 4.1. O item 4.2 descreve o mapeamento do modelo relacional adotado tipicamente por SGBDs para o modelo de componentes, uma simplificação para possibilitar a implementação do protótipo. O item 4.3 descreve a criação do ambiente do componente principal (chamada de $Control$), que implementa um esqueleto do GA e seus respectivos serviços descritos anteriormente. Finalmente, os algoritmos para substituição de componentes propriamente ditos são descritos.

4.1. Modelo de Componentes Fractal

Fractal [Bruneton et al. 2006] é um modelo de componentes modular, extensível com suporte a várias linguagens de programação (Java e C, e de forma experimental para .NET, SmallTalk e Python). Fractal permite projetar, implementar, executar e reconfigurar dinamicamente sistemas e aplicações. Um componente Fractal é um elemento de execução encapsulado, com identificação única e que suporta uma ou mais interfaces. A interface é um ponto de acesso para um componente e que implementa uma interface da linguagem, que representa as operações suportadas.

Uma interface de um componente Fractal pode ser de dois tipos: interface cliente (ou requerida) e servidora (ou provida). Um componente usa a interface cliente para invocar operações implementadas por outros componentes. Um componente usa a interface servidora para invocar operações implementadas pelo próprio componente.

Genericamente, um componente Fractal possui duas camadas, a interna e a externa. A camada externa (ou membrana) possui as interfaces de controle que permitem introspecção e reconfiguração de componentes. A camada interna (ou conteúdo) consiste em um conjunto finito de outros (sub)componentes. As interfaces da membrana podem ser internas (acessíveis apenas pelos subcomponentes) e externas (aquelas acessíveis por outros componentes).

Em Fractal, componentes são conectados a outros componentes por interfaces de vinculação, que podem ser de dois tipos: primitivas ou compostas. Em uma vinculação primitiva, ou em um componente primitivo, uma chamada a uma interface cliente resulta diretamente na chamada da interface vinculada de servidor. Na vinculação composta, ou componente composto, a chamada envolve um ou mais componentes pois as interfaces de cliente e servidor não combinam (e é necessário executar algum tipo de conversão) ou porque os componentes conectados são hospedados em máquinas distintas.

Componentes tem controladores que são usados para acessar operações internas (do próprio componente) e operações externas (em outros componentes). Existem quatro tipos de controles em Fractal:

- **Ciclo de vida**: possibilita a reconfiguração dinâmica dos componentes, pois trata explicitamente da disponibilidade dos mesmos. Há duas operações disponíveis: *startFc* e *stopFc*. Basicamente, *startFc* ativa o componente e *stopFc* pára o componente.
- **Vinculação**: gerencia as dependências entre os componentes. Para realizar a remoção de um componente do sistema, necessariamente, o controle de vinculação deve ser utilizado a fim de promover a desconexão entre os componentes, e a futura religação entre eles.
- **Conteúdo**: realiza o controle de conteúdo para permitir da adição e remoção de subcomponentes. As operações disponíveis neste controle são *addFcSubComponent* e *removeFcSubComponent*.
- **Atributo**: controla atributos para configurar propriedades nos componentes. Tipicamente, atributos são tipos primitivos utilizados para configurar o estado de um componente.

Neste contexto, uma ADS pode ser vista como uma coleção de operações efetuadas pelos controladores de Fractal. Para que a ADS ocorra, um componente deve ser parado pelo controlador do ciclo de vida. Então, o controlador de vinculações trata dependências entre os componentes. Em seguida, o controlador de conteúdo remove a versão antiga e instala a nova versão do componente. Finalmente, o componente é reativado pelo controlador do ciclo de vida.

4.2. Mapeamento do Modelo Relacional para o Modelo de Componentes

Para realizar a substituição de componentes via Fractal, componentes do SGBD precisam ser descritos pelo modelo de componentes Fractal. Como a implementação de um SGBD de acordo com o modelo de componentes não existe, foi realizado um mapeamento dos módulos de um SGDB para componentes caixa-preta. A implementação sobrescreveu de forma mais fiel possível a estrutura básica de um SGBD. A classe raiz implementada pelo protótipo do SGBD é $Componente$, onde todos os subcomponentes do SGBD serão derivados. A classe $SgbdEmComponentesFractal$ funciona como porta

de entrada das requisições ao SGBD. As classes $Parser$, $Optimizer$, $ExecutionPlan$, $OperatorEvaluator$, $ExecutionEngine$, $TransactionManager$, $RecoveryManager$, $LockManager$, $FileAccessAndMethods$, $BufferManager$ e $StorageManager$ simulam os módulos componentes do SGBD.

Uma outra limitação deste procedimento sobre a tecnologia atual é a impossibilidade de um componente Fractal operar como servidor para mais de um componente. Para contornar esta restrição, a estrutura básica do SGBD sofreu uma simplificação do modelo, usando sequencialização das atividades, ou seja, um componente somente requisita serviço de um componente e provê serviço para um componente de cada vez.

4.3. Criação do Ambiente em Fractal

$Control$ é a implementação simplificada do GA. A classe $Control$ usa o método $getBootstrapComponent$ para criar o componente raiz que coordena todo o ambiente. A fábrica de tipos é criada pelo método $getTypeFactory$ e os tipos são criados com o método $createFcType$. A fábrica dos demais componentes é criada com o método $getGenericFactor$. Com o método $newFcInstance$ são criadas novas instâncias de componentes Fractal. A composição dos componentes (ou adição de um subcomponente a um componente composto) é realizada com o método $addFcSubComponent$. A vinculação dos subcomponentes entre si é possibilitada pelo método $bindFc$. O ambiente criado é ativado com o método $startFc$.

4.4. Algoritmos para Atualização de um Subcomponente

O processo de substituição de um (sub)componente é descrito por dois algoritmos. O Algoritmo 1 descreve o modo de operação do GA e o Algoritmo 2 descreve a substituição de componentes propriamente dita.

A substituição de componentes é uma operação permanente, ou seja, enquanto o sistema estiver em operação, substituições de subcomponentes poderão ocorrer. Resumidamente, o Algoritmo 1 executado pelo GA verifica continuamente a fila de novos componentes (linha 2). A partir da existência de um novo componente nesta fila, a substituição de fato do subcomponente é iniciada. O novo componente então é identificado e seu tipo é validado (linha 5): o novo componente deve ser mais recente que o atual (via **Serviço Gerenciador de Versão**) e deve implementar todas as operações da interface do antigo componente (via **Serviço Gerenciador de Integridade**, linha 9).

Na sequência ocorre a decisão sobre a janela de tempo adequada para ocorrer a atualização (via **Serviço Gerenciador de Janela de Tempo**, linha 11). Como já mencionado anteriormente, a substituição do subcomponente somente acontece se o impacto sobre o sistema não interferir demasiadamente na qualidade do serviço. Em períodos de maior carga do sistema, a operação de atualização deve ser evitada. Para avaliação da carga do sistema, o GA utiliza informações como número de requisições ativas e o custo destas requisições para decidir pela substituição do subcomponente.

O ato de substituir um subcomponente começa pela parada do componente raiz, de acordo com Algoritmo 2 (linha 1). A partir deste instante, todos os subcomponentes não executarão mais as suas funções especificadas. Requisições recebidas após a parada de um componente são tratadas pelo **Serviço Gerenciador de Requisições** e executadas

Algoritmo 1: Serviços executados pelo GA

```
1: while true do
2:     Ler componente da fila_de_novos_componentes
3:     if Existe componente then
4:         Ler tipo do componente
5:         if Tipo do componente é valido then
6:             Envia componente para Serviço Gerenciador de Versão
7:             if O componente é mais novo que componente instalado then
8:                 Validar interface do componente via Serviço Gerenciador de Integridade
9:                 if O novo componente tem a interface válida then
10:                    Solicita janela para Serviço Gerenciador de Janela de Tempo
11:                    if Sistema pode ser atualizado then
12:                        Serviço Gerenciador de Requisição substitui componente
                           { executa Algoritmo 2 }
13:                    end if
14:                end if
15:            end if
16:        end if
17:    end if
18: end while
```

ao término deste processo. Segue-se então a desconexão de componentes e subcomponentes (linha 2). Então, o componente é removido (linha 3), ou seja, deixa de ser um subcomponente da arquitetura. O novo componente então é adicionado (linha 4). Neste instante ainda não está apto a funcionar, pois precisar ser informado sobre suas vinculações. Conectado o novo componente, então o componente raiz pode ser iniciado (linha 6). Faltam apenas operações de controle como a atualização da tabela de controle das versões (via **Serviço Gerenciador de Versão**) e a movimentação do componente para o *log* de componentes (linhas 7 e 8).

Algoritmo 2: Substituição de componente

```
1: stopFc root { parar componente raiz }
2: unbindFc component { desvincular subcomponente a ser substituído }
3: removeFcSubComponent { remover subcomponente }
4: addFcSubComponent { adicionar novo subcomponente }
5: bindFc component { vincular novo subcomponente dos demais }
6: startFc root { ativar componente raiz }
7: Mover componente antigo para log
8: Atualizar tabela de versões
```

5. Avaliação Experimental

Para avaliar a possibilidade da atualização de SGBD através do modelo de componentes de *software*, foi executado um conjunto de testes sobre um protótipo que simula o esqueleto do GA envolvendo diferentes cenários e diferentes sobrecargas. A avaliação experimental objetivou estimar o custo da substituição de componentes em um protótipo que executa a substituição automática de componentes de um SGDB durante a execução do serviço para os clientes.

Testes foram realizados em uma máquina virtual *Oracle Virtual Box* 4.0.4, com sistema operacional *Linux Ubuntu* 10.10, sendo a memória base de 512 MB. Este ambi-

ente executa sobre uma máquina real com processador *Intel Core 2 Duo T8300*, 2 GB de memória RAM e sistema operacional *Windows XP Professional* 2002 SP 2.

A avaliação considerou três cenários distintos: (i) **Sem Fractal**, ou seja, execução de protótipo de SGBD construído de forma tradicional (orientado a objetos), portanto sem Fractal (para avaliar o sistema base, isto é, o servidor de banco de dados sem atualização de componentes, (ii) **Fractal sem ADS** , ou seja, execução de protótipo de SGBD construído na perspectiva de componentes de *software* baseado em Fractal, mas sem a realização de ADS, para avaliar o custo do serviço Fractal, e (iii) **Fractal com ADS**, ou seja, execução de um protótipo de SGBD construído na perspectiva de componentes de *software* baseado em Fractal, mas com a realização de ADS, para avaliar o custo de Fractal, com a substituição de componentes.

Os testes realizados consistiram da execução de grupos de requisições de clientes e obtenção do número de transações processadas por segundo. Os grupos definidos para os testes foram 100, 500, 1.000 e 3.000 requisições emitidas, respectivamente. O ambiente de testes não suportou maior quantidade de requisições. Ocorre estouro de memória (restrição de *hardware*).

A Tabela 1 apresenta resultados obtidos com os experimentos indicando o número de requisições emitidas e processadas por segundo em cada um dos três cenários distintos. Para o cálculo dos valores apresentados na referida tabela, foi considerado o tempo médio de processamento das execuções.

Quantidade de requisições emitidas	**Quantidade de requisições processadas**		
	Sem Fractal	Fractal sem ADS	Fractal com ADS
100	17,10	13,47	13,76
500	60,47	43,56	43,65
1.000	92,09	57,28	59,52
3.000	119,10	85,11	80,93

Tabela 1. Comparativo do número de requisições processadas em 1s

O que se pode ser observado na Tabela 1, com a execução desta avaliação experimental é que, como esperado, o próprio modelo de componentes de *software* ainda é um entrave para o desempenho de sistemas distribuídos: de acordo com restrições de implementação do modelo Fractal, os componentes são modelados em série, obedecendo uma ordem pré-estabelecida. Adicionalmente, substituição sem parada de componente não é permitida. A maior perda de desempenho não ocorreu durante a realização da atualização de componentes propriamente dita, mas do fato do SGDB ser modelado como diferentes componentes em série. Note que, na Tabela 1, os valores obtidos para as colunas **Fractal sem ADS** e **Fractal com ADS** (por exemplo 85,11 e 80,93 na última linha da tabela) são bem parecidos, enquanto que os valores obtidos na coluna 1 (**Sem Fractal**), sem a interferência do modelo de componentes apresenta valores mais altos (por exemplo, 119,10 na última linha da tabela), portanto indica o melhor desempenho (isto é, maior quatidade de transações sendo executadas na unidade de tempo).

6. Conclusões e Trabalhos Futuros

ADS não é um assunto novo, mas na prática soluções automáticas são restritas. Desde que a demanda de aplicações críticas que exigem disponibilidade contínua, principal-

mente aquelas que fazem uso de bancos de dados através da Internet, tem aumentado, é interessante que sistemas computacionais sejam providos de técnicas de baixo custo, robustas e transparentes para a atualização de *software* sem a parada do serviço.

Este trabalho apresentou uma arquitetura para construção de um SGBD, baseada em componentes de *software* capaz de permitir a sua atualização automática e transparente, sem a indisponibilização total do sistema e sem usar *hardware* redundante. Propriedades adicionais incluem a ausência de restrição de linguagem e ambiente, flexibilizando portabilidade para diferentes ambientes operacionais e linguagens, oportunizada pelo modelo de componentes, e facilidade de manutenção à medida que a evolução de *software* ocorre de forma contínua.

Foi realizada avaliação experimental através da implementação de um protótipo. A avaliação experimental demonstrou que a implementação da ADS em SGBDs com suporte do modelo de componentes é viável, embora existam restrições impostas pelo próprio modelo de componentes e pela tecnologia atual. A degradação média do desempenho obtida nos testes realizados em ambiente controlado foi de aproximadamente 28,78%, em relação ao protótipo construído no paradigma de orientação a objetos, demonstram que o *framework* Fractal provoca uma sobrecarga significativa no sistema. Comparando os resultados obtidos nos testes sem realização de ADS e realizando ADS, observa-se que a substituição de componentes não provoca quedas adicionais de desempenho. Isto significa que o custo de Fractal é único, independente de quanto se utiliza suas funcionalidades.

Trabalhos futuros incluem a implementação da solução aqui proposta com outros *frameworks* de suporte ao desenvolvimento baseado em componentes de *software*. Esta sugestão tem como objetivo verificar a possibilidade de minimizar a degradação média de desempenho aferida neste trabalho. Outro trabalho futuro é ampliar a arquitetura proposta para suportar SGBDs executados em múltiplos nós. Para sistemas nesta estrutura, a atualização deve acontecer em cada um dos nós de forma coordenada. O desafio consiste em gerenciar esta atualização, a fim de determinar quando todos os nós estarão atualizados, visando atualização completa de um $cluster$ ou de uma $cloud$. Outro desafio é a observação do comportamento da solução aqui proposta através de uma implementação (real) prática. A implementação real possibilitará refinar a solução proposta, identificando pontos críticos e apontando novas direções de pesquisa. É necessário observar também o grau de alterações que serão necessárias na conversão de um SGBD desenvolvido no paradigma de orientação a objetos para o modelo de componentes de *software*.

Finalmente, duas funcionalidades já implementadas pelo GA podem ser melhoradas. Uma delas é o serviço de escolha de janela de tempo para realização da atualiação, que pode ser aprimorado com a adição de mecanismos de previsão de multidões [Baryshnikov et al. 2005], evitado assim, que atualizações sejam realizadas quando a base de dados estará sujeita a demandas excessivas. Outro módulo que pode ser melhorado é o esquema de controle de versão, que pode adicionar ao GA a possibilidade de reverter uma atualização. Para isto, além de guardar a versão antiga do componentes em uma base de dados, um *log* precisará armazenar todas as operações realizadas, ou pelo menos a última operação realizada. A reversão consiste em aplicar o *log* de baixo para cima, fazendo com que a versão antiga de um componente sobrescreva uma versão atual.

Referências

Baryshnikov, Y.; Coffman, E. G.; Pierre, G.; Rubenstein, D.; Squillante, M.; Yimwadsana, T. (2005). Predictability of Web-server traffic congestion, *10th International Workshop on Web Content Caching and Distribution (WCW 2005)*, Sophia Antipolis, France.

Bass, L.; Clements, P.; Kazman, R. (2003). *Software architecture in practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, second edition.

Bruneton, E.; Coupaye, T.; Leclercq, M.; Quema, V.; Stefani, J. B. (2006). The Fractal component model and its support in Java: experiences with auto-adaptive and reconfigurable systems. *Software Pract. Exper.*, 36(11-12):1257–1284.

Elmasri, R.; Navathe, S. (2005). *Sistemas de banco de dados*. Pearson Addison Wesley, São Paulo, 4 edição.

Fabry, R. S. (1976). How to design a system in which modules can be changed on the fly. In *Proceedings of the 2nd International Conference on Software Engineering*, ICSE '76, pp. 470–476, Los Alamitos, CA, USA, IEEE Computer Society Press.

Leger, M.; Ledoux, T.; Coupaye, T. (2007). Reliable dynamic reconfigurations in the Fractal component model. In *Proceedings of the 6th International Workshop on Adaptive and Reflective Middleware: held at the ACM/IFIP/USENIX International Middleware Conference*, ARM '07, pp. 3:1–3:6, New York, NY, USA, ACM.

Lyu, J.; Kim, Y.; Kim, Y.; Lee, I. (2001). A procedure-based dynamic software update. *Proceedings of the 2001 International Conference on Dependable Systems and Networks (formerly: FTCS)*, DSN '01, pp. 271–284, Washington, DC, USA. IEEE Computer Society.

Oreizy, P.; Medvidovic, N.; Taylor, R. N. (1998). Architecture-based runtime software evolution. *Proceedings of the 20th International Conference on Software Engineering*, ICSE '98, pp. 177–186, Washington, DC, USA, 1998. IEEE Computer Society.

Oreizy, P.; Gorlick, M. M.; Taylor, R. N.; Heimbigner, D.; Johnson, G.; Medvidovic, N.; Quilici, A.; Rosenblum, D. S.; Wolf, A. L. (1999). An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14:54–62.

Ramakrishnam, R.; Gehrke, J. (2003). *Database management systems*. McGraw-Hill Education, Singapura, 3 edition.

Segal, M.; Frieder, O. (1993). On-the-fly program modification: systems for dynamic updating. *Software, IEEE*, 10(2):53–65.

Stoyle, G.; Hicks, M.; Bierman, G.; Sewell, P. e Neamtiu, I. (2007). Mutatis mutandis: Safe and predictable dynamic software updating. *ACM Transaction Program. Languages and Systems*, 29(4).

Wahler, M.; Richter, S.; Oriol, M. (2009). Dynamic software updates for real-time systems. *Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades*, HotSWUp '09, pp. 2:1–2:6, New York, NY, USA. ACM.

Wang, Q.; Shen, J.; Wang, X.; Mei, H. (2006). A component-based approach to online software evolution: Research articles. *J. Software Maint. Evolution*, 18:181–205.

Assistente para Desenvolvimento de Software Crítico segundo a IEC 61508

Diego Bandeira, Taisy S. Weber, Sergio L. Cechin, Rodrigo Dobler, João C. Netto

Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil
(taisy,dcbandeira,cechin,rjdobler,netto)@inf.ufrgs.br

Abstract. *An assistance tool can ease the development of critical software when it must follow a given safety standard as the IEC 61508. The standards are extensive and detailed making it a hard reading for software developers unfamiliar with the area of functional safety and fault tolerance techniques. We implemented a tool to help developers and testers to understand and apply each safety requirement of the IEC 61508 concerning the software life cycle. The tool verifies if the developers completed all the activities of a given phase of the life cycle and helps to keep the documentation needed for the certification process.*

Resumo. *Uma ferramenta de apoio ao projeto de software facilita o desenvolvimento de software seguro principalmente quando este deve seguir normas rigorosas de segurança crítica como a EC 61508. A norma é extensa e detalhada tornando penosa sua aplicação por desenvolvedores não familiarizados com as áreas de tolerância a falhas e segurança funcional. Apresentamos uma ferramenta de apoio para facilitar a aplicação de cada um dos requisitos de segurança da IEC 61508 levando em consideração o ciclo de vida de software crítico. A ferramenta verifica se todas as atividades foram completadas e mantém a documentação necessária ao processo de certificação de segurança.*

1 Introdução

Na aplicação de recursos computacionais para controle, automação e monitoramento na indústria nuclear e química, na exploração de petróleo e gás, na instrumentação médica e nos transportes, a necessidade de que os equipamentos programáveis executem corretamente as funções de segurança é evidente. Falhas no software desses equipamentos podem trazer prejuízos irremediáveis para a qualidade de vida ou danos irrecuperáveis ao meio ambiente. O termo segurança (*safety*) aplicado no contexto de prevenção de acidentes não deve ser confundido com o seu outro significado (*security*), que envolve proteção contra intrusos maliciosos, confidencialidade e integridade [1].

Exemplos de funções de segurança são: sinalização em ferrovias; controle de parada emergencial em indústrias químicas; bloqueio preventivo em maquinários pesados; sinais luminosos de alerta; sistemas de antitravamento de freios em automóveis; alarmes de incêndio e detecção de gases tóxicos. Funções de segurança costumavam ser implementadas com componentes discretos, não programáveis. Entretanto, é grande a demanda pelo emprego de componentes programáveis, tais como controladores lógicos e microcontroladores, com a função de segurança sendo executada por

Projeto parcialmente financiado pela Rede E3 – RIO-SIL/FINEP 01.11.0154.00

software [2]. As vantagens são inúmeras: reuso, maior flexibilidade, adaptabilidade e desempenho, além de um menor custo de desenvolvimento e manutenção.

Há muitas diferenças entre o processo de desenvolvimento de software em projetos tradicionais e em projetos de sistemas seguros. Nesses últimos, a criatividade do desenvolvedor é condicionada à aplicação de técnicas que já se mostraram adequadas na prática. Inovações tecnológicas não são bem acolhidas pelas normas de segurança, como a IEC 61508 [3], devido à falta de evidências sobre sua efetividade em situações reais de perigo. O ambiente de operação de sistemas relacionados à segurança é geralmente hostil. Fatores como temperatura, pressão, umidade, salinidade, gazes corrosivos, trepidação e impacto impõem um maior desgaste aos equipamentos, reduzindo sua vida útil e aumentando a probabilidade de apresentarem defeitos.

O maior desafio para um desenvolvedor de software seguro é seguir criteriosamente a regulamentação aplicável, escolhendo em cada fase do desenvolvimento as técnicas mais apropriadas para o domínio de aplicação. As normas são extensas e detalhadas e sua leitura e interpretação são difíceis. Para agilizar a sua aplicação, um software assistente passa a ser uma ferramenta essencial ao desenvolvedor [4]. A ferramenta direciona o desenvolvimento de maneira que nenhuma atividade ou documento importante seja ignorado. Ela acompanha passo a passo as tarefas do desenvolvedor e indica todas as atividades necessárias em cada fase do ciclo de desenvolvimento. Uma boa ferramenta mantém o histórico do projeto e facilita o rastreamento de alterações realizadas durante o período de desenvolvimento.

O objetivo do trabalho é implementar uma ferramenta de apoio para facilitar a aplicação da norma internacional IEC 61508 no desenvolvimento de software para a área de segurança funcional crítica. A ferramenta será aplicada no projeto RIO-SIL, que visa à produção de módulos com portas de entrada e saída digitais para sistemas instrumentados de segurança. Esses módulos são equipamentos completos de hardware e software conectados, por um lado, a barramentos de campo para comunicação com um ou mais controladores centrais e, do outro lado, conectados ao processo sendo instrumentado em tempo real. A ferramenta servirá não apenas aos propósitos do projeto RIO-SIL, mas terá um papel fundamental no treinamento da equipe de desenvolvimento. Será usada também como recurso de aprendizagem nas áreas de confiabilidade e tolerância a falhas para sistemas críticos.

A ferramenta não é um sistema crítico, nem executa funções de segurança. Por esse motivo seu desenvolvimento não está condicionado às restrições da norma IEC 61508. A ferramenta também não gera código seguro, nem executa os testes recomendados pela norma. A ferramenta apenas assiste na aplicação da norma, indicando ao desenvolvedor, a cada passo, as técnicas e métodos recomendados e mantendo em uma base de dados todos os documentos gerados necessários para a avaliação de um dado projeto.

O artigo apresenta o protótipo da ferramenta, focando nas fases relacionadas ao ciclo de vida de projeto de software. Inicialmente apresenta-se um resumo da norma IEC 61508 e a metodologia sugerida pela norma para o desenvolvimento de equipamentos seguros. Segue então um breve resumo de algumas ferramentas comerciais mais citadas e a apresentação ferramenta assistente de segurança, SVA, proposta. O artigo conclui com alguns comentários sobre a experiência na aplicação da norma e da ferramenta.

2 Padrão Internacional para Integridade de Segurança

Equipamentos relacionados à segurança atuam em ambientes onde acidentes podem provocar danos a pessoas ou ao meio ambiente. Tais equipamentos devem garantir confiabilidade, disponibilidade e integridade de segurança. Devem operar corretamente mesmo na ocorrência de falhas de hardware e software e de interferência externa. Caso não seja possível em alguma situação garantir a operação correta, o equipamento deve descontinuar sua operação alcançando um estado seguro sem provocar acidentes.

Normas, como a IEC 61508 [5], preconizam o uso de técnicas de prevenção e de tolerância a falhas para a redução de risco, além do projeto criterioso, documentado e auditável, tanto do hardware como de software, desde as primeiras fases do ciclo de desenvolvimento. O projeto de equipamentos relacionados à segurança exige sofisticado grau de elaboração e inúmeros cuidados técnicos. Além dos aspectos de engenharia de hardware e software, é preciso controlar também os fatores operacionais, os fatores gerenciais, o ambiente de operação e a manutenção. A norma IEC 61508 foi publicada originalmente em 2000. Em 2002 iniciou o processo de revisão [3]. Em abril de 2010 foi lançada a sua segunda edição, com base na qual este trabalho foi desenvolvido.

2.1 IEC 61508

A IEC 61508 [5] é um padrão para equipamentos elétricos, eletrônicos e eletrônicos programáveis relacionados à segurança, independente do seu domínio de aplicação, sejam processos industriais, máquinas, transportes ou energia. A IEC 61508 é amplamente aceita como a melhor norma genérica para gerenciamento de segurança funcional [7].

A norma define um sistema seguro como “livre de riscos inaceitáveis, envolvendo prejuízos físicos ou danos à saúde de pessoas, resultantes direta ou indiretamente de danos à propriedade ou ao ambiente” [7]. Ela apresenta uma abordagem genérica para todas as fases e atividades do ciclo de vida e um conjunto de técnicas e métodos para o desenvolvimento de equipamentos usados para realizar funções de segurança. A norma inclui também os procedimentos técnicos e administrativos necessários para atingir a integridade de segurança funcional desejada.

A norma descreve requisitos específicos para o desenvolvimento do hardware e do software para equipamentos eletrônicos programáveis [8], o que representa uma inovação em relação a normas anteriores, que inibiam o uso de sistemas programáveis devido aos riscos inerentes relacionados ao software.

2.2 Integridade de Segurança

Integridade de segurança é um conceito associado à probabilidade que a função de segurança seja executada satisfatoriamente e é indicada por um nível discreto chamado SIL. Os níveis previstos para SIL são 1, 2, 3 e 4, sendo que SIL 4 representa a maior integridade de segurança prevista pela norma. As exigências de projeto crescem do nível 1 ao 4.

Para hardware, a norma relaciona o SIL à taxa de defeitos perigosos resultantes de falhas não cobertas pelos mecanismos de detecção empregados e a restrições na arquitetura. O cálculo é realizado dispondo da taxa de defeitos de todos os componentes de hardware usados, da árvore de propagação de falhas, da distinção entre falhas

seguras e perigosas, do reconhecimento de falhas de modo comum, do grau de redundância da arquitetura e da cobertura dos mecanismos de detecção de falhas empregado, além de várias outras informações técnicas [8]. Se algum dos componentes é programável, a determinação do SIL alcançável deve incluir também a análise e verificação detalhada do software que executa no componente.

Todas as técnicas e estratégias usadas para o desenvolvimento e teste do hardware e do software do equipamento também são analisadas para determinar se um projeto tem condições de alcançar o SIL almejado. Essa análise visa determinar a robustez do equipamento a falhas sistemáticas, que são típicas de software e de projeto. Falhas sistemáticas diferem de falhas randômicas, que são típicas dos componentes de hardware. Para essa análise, a norma apresenta técnicas e medidas associadas às fases do ciclo de vida do equipamento, que vão desde a especificação até a retirada de operação e descarte [9]. Técnicas e medidas são classificadas como não-recomendadas (NR), recomendadas (R), altamente recomendadas (HR) ou não há recomendações contra ou a favor (--). A norma apresenta técnicas alternativas ou equivalentes, que devem ser escolhidas de acordo com sua adequação à aplicação. Para cada nível de SIL almejado, técnicas específicas são sugeridas ou impostas.

2.3 Certificação

É comum órgãos reguladores exigirem que equipamentos que executam funções de segurança sejam certificados segundo normas apropriadas. O objetivo de uma certificação de segurança é prover a garantia, reconhecida pelo órgão regulador, que o sistema foi considerado seguro pelo órgão certificador [7]. O conjunto de técnicas e medidas adotadas em todas as fases do ciclo de vida deve convencer o órgão certificador que o projeto apresenta o nível de integridade de segurança solicitado. Para o hardware são usados valores numéricos como taxa de defeito dos componentes e cobertura das técnicas de detecção e diagnóstico. No caso do software, não são usados valores numéricos: uma implementação deve seguir rigorosamente todas as recomendações da norma para o SIL desejado [11]. Considerando-se o SIL do hardware e do software, o menor deles será o SIL do sistema.

Quando uma recomendação não é seguida, é contrariada ou quando se usam alternativas não previstas, a certificação ainda é possível, mas a norma exige uma forte fundamentação baseada em argumentos técnicos [10]. Essa argumentação pode colocar um fator de incerteza na certificação, pois não é possível saber a priori se a argumentação será aceita pelo órgão certificador [11].

A função de segurança é certificada, principalmente, através da documentação gerada durante todas as fases do ciclo de vida do seu desenvolvimento. Cuidados devem ser tomados para definir documentos apropriados à certificação, mas a norma não estabelece o formato dos documentos. A certificação não é restrita ao produto gerado pelo desenvolvimento. Como a norma se aplica ao ciclo de vida, todo o processo de desenvolvimento e produção é levado em consideração para a certificação de segurança.

3 Metodologia de Desenvolvimento de Sistemas Seguros

O desenvolvimento dos equipamentos relacionados à segurança deve garantir a máxima cobertura de detecção de falhas randômicas possível para o SIL desejado. Deve também incluir medidas de prevenção de falhas. Todo o processo de desenvolvimento deve

seguir uma metodologia que inicia na especificação de segurança do sistema e contempla todas as etapas de implementação, verificação e validação. Testes devem acompanhar todas as etapas, da especificação à operação do equipamento em campo. A norma enfatiza a necessidade de planejar, especificar, desenvolver, testar e documentar os aspectos relativos à segurança funcional. A norma também sugere e, em alguns casos, impõe como essas atividades devem ser realizadas para alcançar um sistema com determinado nível de integridade de segurança.

A IEC 61508 é extensa, com 7 volumes, inúmeros anexos e centenas de páginas, textual na sua maior parte, e não convenientemente estruturada [7]. É fácil para um desenvolvedor se confundir com as inúmeras definições, técnicas e medidas impostas ou sugeridas, fases de projeto e documentação necessária. Neste contexto, deve ser considerado o emprego de uma ferramenta que incorpore, de forma estruturada, o conhecimento necessário para a aplicação da norma.

3.1 Tolerância a Falhas e a Norma IEC 61508

O desenvolvimento de hardware e software deve incorporar métodos e técnicas da área de tolerância a falhas. Para isso, recorre-se ao emprego de redundância de componentes, diagnóstico e detecção de falhas, diversidade de componentes de hardware e de software, degradação suave e à aplicação de uma série de proteções extras, de maneira a atingir um estado seguro no caso de ocorrência de falhas. Entretanto, nem todas as técnicas usuais de tolerância a falhas são recomendadas, e algumas são explicitamente não-recomendadas. Por exemplo, a técnica popular de recuperação para um estado anterior da computação não é recomendada para SIL 4. Recuperação para um novo estado posterior a falha, que aparecia como técnica recomendada em 2000, não é mais citada na edição de 2010. Correção de falhas usando inteligência artificial não é recomendada de SIL 2 a 4, sendo indiferente para SIL 1. Assim não basta o bom senso e a experiência do desenvolvedor: a norma deve ser conferida a cada escolha de técnica ou medida a ser empregada.

A norma assume que falhas de hardware e software são inevitáveis. O erro causado pela falha deve ser detectado e o sistema deve corrigir o erro ou ir para um estado seguro. Mas é impossível detectar todos os erros. A taxa residual de defeitos, devido à cobertura imperfeita dos mecanismos de tolerância a falhas, detecção e diagnóstico empregados irá, em última análise, determinar o nível de integridade de segurança do equipamento; quanto menor essa taxa, maior o SIL.

Sistemas com menor probabilidade de apresentar defeitos residuais da função de segurança são classificados em SIL 4 (10^{-9} defeitos perigosos por hora ou 10^{-5} defeitos sob demanda em baixa demanda de operação). Aumentando uma ordem de grandeza na taxa de defeitos, diminui de uma unidade o valor do SIL, até SIL 1. Por exemplo, uma função de segurança de detecção de incêndio que apresente, no máximo, um defeito a cada 10.000 demandas, seria classificada como SIL 3. Um defeito no caso corresponde a não atuar em caso de incêndio.

Vale notar que a norma não altera ou inova em relação a boas práticas de engenharia de software e ao emprego de técnicas de tolerância a falhas. Mas exige comprovação documentada da eficiência de seu emprego em um dado projeto.

3.2 Desenvolvimento de Software Seguro

Boas práticas de projeto não são suficientes para garantir a segurança necessária [6] para sistemas de segurança. Ao contrário dos procedimentos aplicados ao hardware [11], não há mecanismos universalmente aceitos para determinação da taxa de defeitos de software [11]. Na IEC 61508, a ênfase está na prevenção e controle de falhas. Deve ser cuidadosamente controlado o modo como as atividades de desenvolvimento de software são realizadas. Devem ser enfatizadas boas práticas, técnicas e medidas que auxiliem alcançar a integridade de segurança desejada.

Um ciclo de vida voltado à segurança deve incluir procedimentos técnicos e administrativos. Entre os técnicos estão a integração de técnicas e medidas voltadas à segurança e as atividades do ciclo de vida do equipamento. Entre os procedimentos administrativos estão o planejamento da segurança funcional e procedimentos rígidos para a gerência de configuração de software. Os procedimentos administrativos estão relacionados à gerência de qualidade do software.

O planejamento de segurança define a estratégia para a obtenção, o desenvolvimento, a integração, a verificação, a validação e a modificação do software de acordo com o nível da integridade de segurança (SIL) do sistema. Uma função importante da gerência da qualidade é especificar as responsabilidades das pessoas, departamentos e organizações em cada uma das atividades do ciclo de vida de projeto do software [3]. A gerência de qualidade deve também definir procedimentos para acompanhar o sistema em operação e avaliar se a taxa de defeitos está de acordo com o assumido. Para permitir auditoria, além dos documentos relacionados ao projeto todas as atividades de gerência devem também ser documentadas [9].

3.3 Ciclo de Vida para Segurança de Software

Uma das funções da gerência de qualidade de software é definir um ciclo de vida estruturado em fases e atividades. A norma não impõe um ciclo de vida. É enfatizado, entretanto, que procedimentos para garantir qualidade e segurança compatíveis com aqueles contemplados na norma devem ser incorporados nas atividades do ciclo de vida. Cada fase do ciclo de vida deve ser dividida em atividades elementares. Para cada atividade elementar devem ser definidos o seu escopo, as entradas e as saídas. Além disso, cada atividade deve ser documentada.

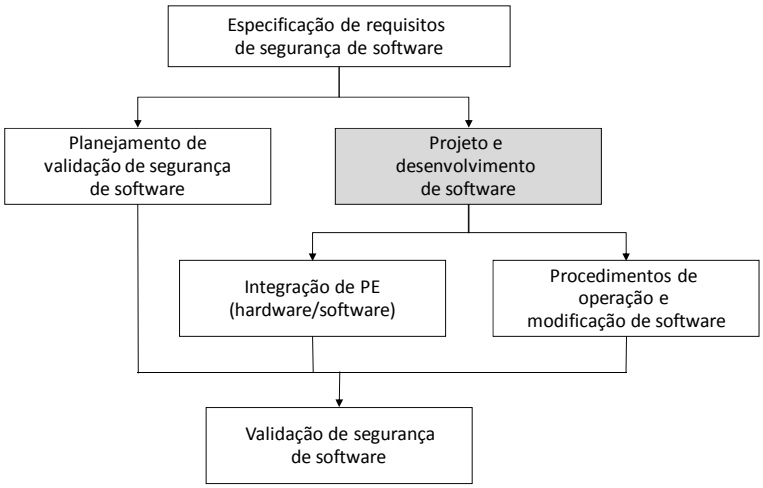


Fig. 1 Ciclo de vida de software seguro

Apesar de não impor, a norma sugere um modelo para ciclo de vida para software (figura 1).

Os requisitos para as funções de segurança e os requisitos para a integridade de segurança são diferentes: os primeiros são derivados da análise de ameaças e indicam o que a função de segurança efetivamente realiza. A integridade de segurança, por outro lado, é a probabilidade que a função de segurança seja executada satisfatoriamente, e os requisitos são derivados da estimativa de riscos. Deve ser enfatizado que as atividades da figura 1 podem fazer parte do ciclo de vida de qualquer bom projeto de software, mesmo aqueles sem necessidade de certificação.

3.4 Projeto e Desenvolvimento de Software

O modelo V (fig 2) definido na IEC 61508 mostra a fase de projeto e desenvolvimento de software do ciclo de vida . O modelo V inicia com a especificação dos requisitos de segurança de software, que é derivada da especificação dos requisitos de segurança global do sistema.

A arquitetura de software é definida a partir da especificação, mas é também influenciada pela arquitetura de hardware do equipamento de segurança. A fase de definição da arquitetura de software estabelece as estratégias de tolerância a falhas e diagnóstico que serão implementadas em software. A fase envolve também a escolha das ferramentas de suporte e linguagens de programação adequadas.

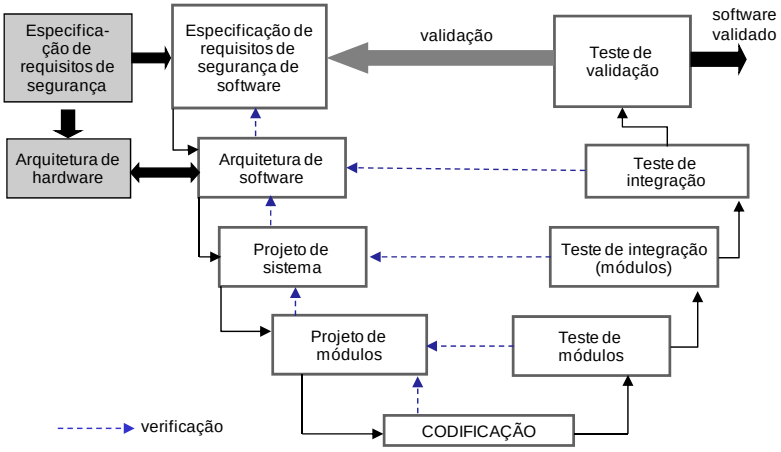


Fig. 2 Modelo V

Validação, verificação e avaliação envolvem técnicas e resultados diferentes. De acordo com a norma, a validação da segurança funcional deve garantir que a integração dos componentes de software e hardware do sistema atenda aos requisitos especificados para a segurança do software no SIL requerido. A verificação de software em relação ao SIL desejado deve testar e avaliar as saídas de uma dada fase do ciclo de vida para garantir correção e consistência em relação às entradas da fase. Finalmente a avaliação da segurança do software deve investigar e julgar a integridade de segurança funcional alcançada pelo sistema.

Para cada uma das atividades do modelo V, a norma associa tabelas que devem ser seguidas. Nessas tabelas constam as técnicas recomendadas para cada nível de integridade de segurança e a documentação que corresponde à atividade. O desenvolvedor deve assinalar nas tabelas todas as técnicas empregadas assim como

justificativas se uma dada técnica recomendada não foi aplicada. A apresentação das tabelas para o desenvolvedor, com a definição dos termos e conceitos, referência à sua localização na norma e espaço para armazenamento e recuperação de documentos, forma a base para ferramentas assistentes de projeto nesta área.

4 Ferramentas de Projeto Visando SIL

Várias ferramentas para auxiliar a aplicação da IEC 61508 estão disponíveis no mercado. Geralmente permitem o cálculo de SIL do hardware e suporte a geração e armazenamento de documentos. Três das mais completas ferramentas são Silcore, exSILentia , e SilSover. Silcore [13] é uma ferramenta da ACM Automation Inc para o projeto e avaliação de sistemas desenvolvidos visando cumprir os requisitos das normas IEC 61508IEC, IEC 61511, e ANSI / ISA 84.0, e é baseada no ciclo de vida do sistema. A ferramenta facilita otimizar o SIL, através do cálculo do intervalo de teste apropriado para o sistema em operação, e dos cálculos de confiabilidade necessários à determinação do SIL do hardware. Para as fases de manutenção e operação, ela oferece opções de rastreamento e geração de relatórios para equipamentos fora de serviço.

ExSILentia [14], da Exida, conta com um módulo para a geração automática de documentação, ficando a cargo do usuário preencher detalhes específicos do projeto. Outros módulos da ferramenta auxiliam na análise de risco e seleção do SIL e ajudam na especificação dos requisitos de segurança. O módulo SILver é a parte responsável pela verificação do SIL, e possui motor de cálculos para SIL de hardware certificado por terceiros. Os dados de confiabilidade dos componentes de hardware podem ser obtidos de uma base de dados. Para as fases finais do ciclo de vida, a ferramenta oferece geração automática de procedimentos para testes e gravação de dados de eventos como testes, falhas e picos de demanda.

SilSolver [15] permite que sejam indicados os dispositivos que compõem o sistema sendo avaliado e que se descreva sua topologia. A partir dessa descrição, a ferramenta avalia o SIL dos componentes de hardware aplicando uma árvore de falhas e informando ao usuário dados como as taxas de defeito do equipamento. A ferramenta permite a visualização da contribuição dos diversos dispositivos para o resultado final da análise, e também fornece apoio à documentação do projeto através de relatórios.

A ferramenta desenvolvida neste trabalho, SVA (Software Validation Assistent), foca no desenvolvimento do software crítico e não no hardware como as demais citadas acima. A principal vantagem da ferramenta em relação às demais é a sua adaptabilidade e facilidade de extensão para outras funcionalidades, inclusive para adaptações a alterações futuras na norma. Deve-se enfatizar que a ferramenta proposta não visa atuar sobre todo o ciclo de vida do sistema, mas apenas nas fases relacionadas ao desenvolvimento de software.

5 SVA: Software Validation Assistant

O objetivo do assistente de validação de software, SVA, cujo protótipo é apresentado neste artigo, é servir como ferramenta no apoio ao projeto de software a ser desenvolvido em conformidade com a norma IEC 61508.

Um assistente é útil para o desenvolvimento de software seguro, pois as técnicas e medidas sugeridas ou impostas pela norma nem sempre correspondem à experiência usual dos desenvolvedores, o que aumenta a complexidade na sua aplicação. As

ferramentas disponíveis no mercado são de alto custo, o que inibe seu uso para aprendizagem e familiarização. Para os desenvolvedores adquirirem habilidades na aplicação da norma, e também para fins de treinamento e aprendizagem, uma ferramenta como o SVA pode ser de grande auxílio.

A garantia de certificação não está assegurada com o uso do SVA ou de qualquer outra ferramenta comercial, mesmo seguindo todos os passos corretamente, uma vez que não existem meios algorítmicos para avaliar determinada documentação. Entretanto, uma ferramenta de apoio guia a equipe de desenvolvimento através do ciclo de vida informando se todas as recomendações e atividades previstas no ciclo foram cumpridas. O cumprimento de todas as exigências e a estruturação facilitada pela ferramenta facilita o processo de certificação.

O SVA é uma ferramenta de apoio que indica para cada fase do ciclo de vida quais as técnicas que devem ser aplicadas, quais os documentos e artefatos devem ser gerados na fase e quem são os responsáveis pela fase. Nesse sentido ela é um guia para a aplicação da norma.

5.1 Implementação do Assistente de Validação

O assistente de validação, SVA, executa em um servidor Apache Tomcat, sendo suas páginas desenvolvidas em JSP (Java Server Pages). Apache Tomcat é um servidor web Java, ou mais especificamente um container de servlets. JSP é uma tecnologia para aplicações web semelhante a ASP e PHP. JSP permite ao desenvolvedor produzir aplicações que acessem o banco de dados, manipulem arquivos no formato texto, capturem informações a partir de formulários e captem informações sobre o visitante e sobre o servidor. O sistema SVA usa uma instância de banco de dados relacional Apache Derby para efetuar o armazenamento e manipulação dos dados.

5.2 Classes do Assistente de Validação

O diagrama de classes é uma representação da estrutura e relações das classes que servem de modelo para objetos e serve como base para a construção de outros elementos da documentação relativa à ferramenta. O sistema SVA (figuras 3 e 4) divide-se entre 11 classes e três enumerações, totalizando 14 classes.

Foram definidos três tipos de usuários para o SVA: o gerente que pode criar novos projetos e designar desenvolvedores para as diferentes fases do ciclo de vida; o especialista na norma, chamado técnico, que pode atualizar o sistema, e os desenvolvedores que implementam as funções de segurança seguindo passo a passo as recomendações da norma.

A classe *Usuario* (fig. 4) representa a abstração dos dados de usuários (gerente, técnico ou desenvolvedor), com os atributos email, nome, perfil e username. *Projeto* (fig. 3) representa os objetos que contém nomeProjeto, nomeEmpresa, silPretendido, gerenteProjeto e responsavelTecnico (sendo os 2 últimos instâncias da classe *Usuario*).

O SIL pretendido irá definir as técnicas e métodos recomendados e não recomentados para um dado projeto em todas as faces do ciclo de vida. O SIL faz parte da especificação da função de segurança e não pode ser alterado. Ao desenvolvedor serão mostradas apenas as informações necessárias para alcançar o SIL pretendido.

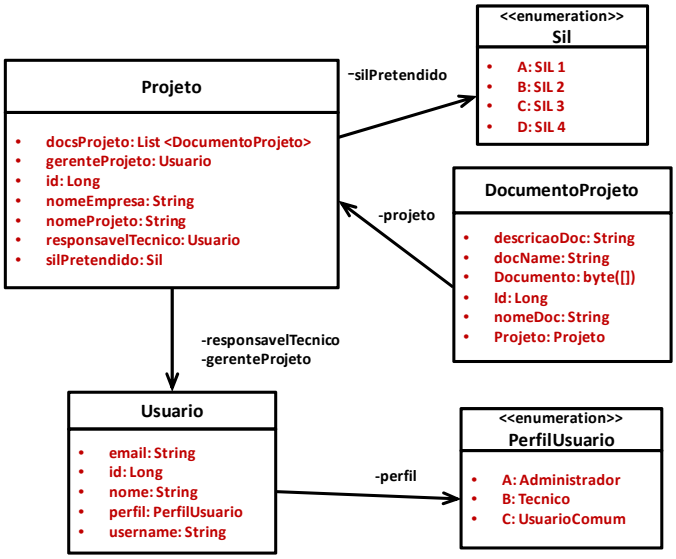


Fig. 3 Diagrama de classes: projeto e usuário

CicloVida (fig. 4) define os atributos dos objetos que representam as fases do ciclo de vida, tais como código, nome, descrição e uma lista de técnicas que são referenciadas. Ao ciclo de vida e ao SIL pretendido estão associadas às técnicas e medidas sugeridas ou impostas pela norma.

A norma organiza as técnicas em tabelas de dois níveis. Várias técnicas gerais do primeiro nível são detalhadas em tabelas do segundo nível. Por exemplo, na fase de especificação dos requisitos de segurança de software (fig 2), a tabela de primeiro nível indica que o uso de métodos semi-formais de especificação é altamente recomendado para SIL 3. A tabela de segundo nível lista os métodos (diagramas funcionais, diagramas de sequência, máquinas de estado finitas, redes de Petri, tabelas de decisão, ...) e seu grau de recomendação. Os dois tipos de tabela são representados por *TecnicaMedidaA* e *TecnicaMedidaB*.

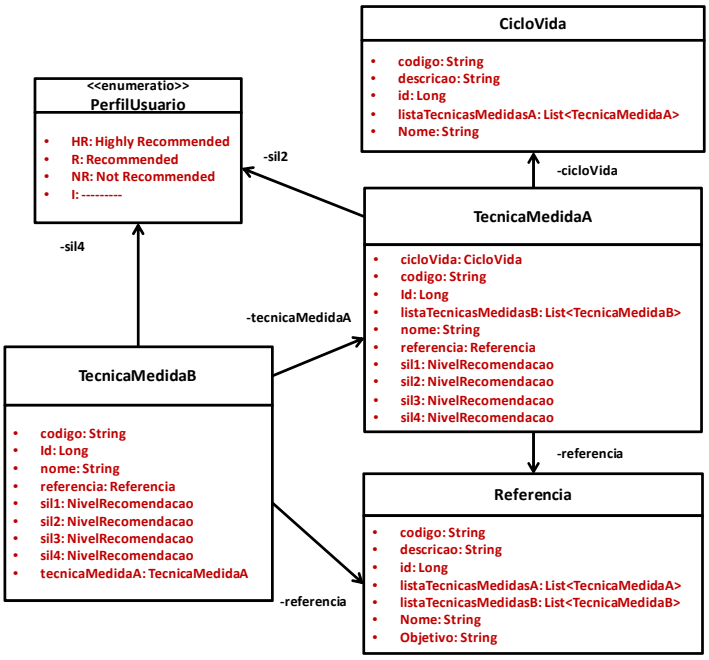


Fig. 4 Diagrama de classes: ciclo de vida

TecnicaMedidaA representa técnicas e medidas através dos atributos codigo, nome, cicloVida (referência a qual fase do ciclo de vida a técnica pertence), sil1, sil2, sil3, sil4, além de uma lista de técnicas das tabelas da norma as quais referencia. *TecnicaMedidaB* por ser um detalhamento das técnicas gerais, liga-se um objeto do tipo *TecnicaMedidaA*. Ambas estão relacionadas à classe *Referencia*, que possui os atributos codigo, nome, descricao e objetivo, além de listas de técnicas (das tabelas dos anexos A ou B da IEC 61503, parte 3) pelas quais são referenciadas.

As classes listadas a seguir completam o assistente e auxiliam na gerência e documentação de um projeto:

- *EventoAgenda*: representa eventos da agenda, com os atributos data, hora, nomeEvento e infoEvento;
- *DocumentoConhecimento*: abstrai objetos que contém o arquivo, o nome dado pelo usuário (nomeDoc), a descrição do arquivo e o nome do arquivo no sistema (docName);
- *LinkConteudo*: representa objetos formados pelo nome, assunto e endereço url;
- *Topico*: abstrai objetos que contém nomeTopico, assuntoTopico e conteudoTopico, no formato String;
- *DocumentoProjeto*: semelhante a *DocumentoConhecimento*, com a adição de um atributo que faz referência a um projeto.
- As enumerações são: *NivelRecomendacao*, *PerfilUsuario*, *Sil*. Seus valores representam um conjunto finito de identificadores previamente definidos.

5.3 Interface com Usuários

A seguir são mostradas algumas das funcionalidades do SVA acessível através da interface com os usuários do sistema. A tela de cadastro de projetos (figura 5) permite cadastrar um projeto com o seu nome, a empresa para a qual está sendo desenvolvido, o SIL pretendido, além do gerente e responsável técnico. Os dois últimos são preenchidos através de uma janela *popup* de busca de usuários já cadastrados, que é mostrada quando do clique no botão "..." do campo desejado.

Fig. 5 Cadastro de projetos

A figura 6 mostra os resultados de uma busca de usuários. O usuário desenvolvedor alocado a um projeto com dada responsabilidade pode agora iniciar o processo de acompanhamento e validação da sua tarefa.

Ao clicar no menu Validação, são apresentados 4 submenus (SIL 1, SIL 2, SIL 3 e SIL 4). Ao selecionar o SIL desejado, o sistema carrega uma tela com o ciclo de vida e as técnicas a ele relacionadas, bem como o índice de referência da técnica e o nível de recomendação para o SIL selecionado.

Fig. 6 Seleção de usuários

As informações sobre as fases do ciclo de vida de segurança de software, bem como suas respectivas técnicas, encontram-se nas tabelas A.1 até A.10 do Anexo A da parte 3 (volume 3) da IEC 61508. O código do ciclo de vida representa a qual tabela de técnicas da norma ele está associado. No caso de uma técnica, representa o próprio código da técnica dentro da tabela da fase do ciclo de vida a qual pertence.

Para efetuar a validação, o usuário seleciona as técnicas que estão sendo utilizadas através da *checkbox* associada a cada técnica. Ao acionar o botão Validar, o sistema verifica se alguma técnica HR (altamente recomendada) não foi marcada, e/ou se alguma técnica NR (não recomendada) foi selecionada. Caso aconteça, o sistema avisa que em ambos os casos há a necessidade de apresentar uma justificativa detalhada que será julgada pelo órgão certificador durante o processo de certificação. A figura 7 mostra a tela com um exemplo de validação para SIL 1.

Fig. 7 Exemplo de tela de validação para SIL 1

Finalmente, no menu Área Técnica estão presentes as funcionalidades que permitem o gerenciamento do ciclo de vida e do modelo V, bem como das técnicas e medidas presentes nas tabelas dos anexos A e B da IEC 61508 parte 3, além das respectivas referências. A área técnica tem por objetivo proporcionar flexibilidade à ferramenta, para torná-la adaptável a eventuais mudanças, como por exemplo, uma nova edição da norma.

5.4 Avaliação da Ferramenta SVA

A ferramenta está sendo utilizada pelos bolsistas e pesquisadores do projeto RIO-SIL, durante as atividades de treinamento do projeto. Esses usuários estão avaliando a ferramenta quando utilizada sob os três papeis que ela disponibiliza: gerente, técnico e desenvolvedor. Algumas funcionalidades, como rastreamento de mudança e geração de

relatórios, ainda estão sendo implementadas. Até o momento, a ferramenta vem cumprindo com seus objetivos de facilitar a compreensão e aplicação da norma IEC 61508 durante treinamento de equipe. Uma avaliação mais completa será possível quando a ferramenta for utilizada não apenas para treinamento, mas para a implementação de um projeto real de uma função de segurança.

6 Conclusão

A IEC 61508 estabelece os requisitos necessários para assegurar que os sistemas sejam projetados, implementados e operados para suprir as exigências do nível de integridade de segurança (SIL) requerido. A norma define também um processo a ser seguido por todas as partes envolvidas, visando uniformizar a terminologia e facilitar a certificação. Embora elogiada por sua adaptabilidade a vários setores, a norma também recebe críticas. A IEC 61508 é deficiente por deixar muitos pontos vagos em relação à documentação e ao tratamento das justificativas quando uma técnica sugerida ou imposta pela norma não é aplicada. Muitas vezes também é necessário recorrer a outras normas para completar os procedimentos necessários para o processo de certificação.

No Brasil, a crescente demanda por equipamentos certificados para segurança funcional faz com o desenvolvimento de software seguro seja igualmente impulsionado. A ferramenta SVA tem o objetivo de auxiliar o desenvolvedor a aplicar técnicas e medidas relacionadas à norma e a manter o registro do caminho percorrido de acordo com o ciclo de vida determinado para o projeto. Tem por objetivo servir como uma alternativa de ferramenta de apoio à aplicação de normas para sistemas relacionadas à segurança, uma vez que as ferramentas existentes não são facilmente disponíveis.

O SVA abrange todas as fases do ciclo de vida de segurança de software, permitindo a inclusão de novas fases e a alteração das fases existentes. Isto é útil para tornar o sistema adaptável a eventuais mudanças na norma, mesmo que estas não ocorram com grande frequência. A mesma facilidade de extensão e alteração é apresentada para as técnicas relacionadas às fases do ciclo de vida, o que proporciona flexibilidade à aplicação. Além de auxiliar na verificação do emprego das técnicas pertinentes, o SVA permite a manutenção de registros de usuários, projetos e eventos. Ela oferece apoio também para a criação de uma base de conhecimento, através de funcionalidades de cadastro e busca de links, tópicos e documentos. Isto possibilita a criação de um banco de informações que sejam relevantes ao contexto e que estejam disponíveis para consulta e acesso rápido pelos usuários do sistema.

No treinamento de desenvolvedores no uso na norma IEC 61508, a prática mostra que a apresentação puramente textual da norma é cansativa [16], o que pode levar a sua aplicação incorreta. Sem o auxílio de uma ferramenta, o tempo de assimilação de todos os detalhes envolvidos na norma IEC, nos seus sete volumes e centenas de páginas é desencorajador. A ferramenta mantém a atenção a cada detalhe do ciclo de vida no momento onde a informação é necessária tornando o desenvolvimento mais produtivo e menos sujeito a erros por omissão ou má interpretação.

7 Referências

[1] Avizienis, A.; Laprie, J.-C.; Randell, B.; Landwehr, C., “Basic concepts and taxonomy of dependable and secure computing.” *Dependable and Secure Computing, IEEE Transactions on* , vol.1, no.1, pp. 11- 33, Jan.-March 2004

[2] Dunn, W. R., “Designing safety-critical computer systems.” *IEEE Comp*, 36(11):40 – 46. 2003.

[3] Bell, R., “Introduction and Revision of IEC 61508”. *Advances in Systems Safety*, 2011,Springer

[4] Faller, R., “Project experience with IEC 61508 and its consequences.” SAFECOMP 2001, v. 2187 of Lecture Notes in Computer Science, pp 200 – 214.

[5] *International Electrotechnical Commission IEC 61508, part 1 to 7; Functional Safety of Electrical, Electronic and Programmable Electronic Safety-Related Systems*. IEC Std. 2010. http://www.iec.ch/functionalsafety.

[6] Johnson, C., “Using IEC 61508 to guide the investigation of computer-related incidents and accidents”. In SAFECOMP 2003, v. 2788 of Lecture Notes in Computer Science, pages 410 – 424.

[7] Panesar-Walawege, R.K. et all. “Characterizing the Chain of Evidence for Software Safety Cases: A Conceptual Model Based on the IEC 61508 Standard”, in *2010 Third International Conference on Software Testing, Verification and Validation*. 335-344

[8] Smith D.J.; Simpson, K.G.L.; *Functional Safety: a straightforward guide to applying IEC 61508 and related standards*, Elsevier, Butterworth-Heinemann, U.K. 2ªedição, 2004.

[9] Brown, S. “Overview of IEC 61508 Design of electrical/electronic/programmable electronic safety-related systems”. *IEEE Computing and Control, Engineering Journal*, fev. 2000.

[10] Mcdermid, J.A. “Software Safety: Where’s the Evidence?”, in *Proc. 6th Australian Workshop on Industrial Experience*, v.3, 2001.

[11] Cechin, Sergio Luis; Weber, Taisy Silva; Netto, Joao Cesar. Arquiteturas MooN(D) para portas de entrada e saída de remotas em conformidade com a IEC 61508 . In: *Congresso Brasileiro de Automática* (19. : 2012 set. 02-06 : Campina Grande, PB). Campinas, SP : Sociedade Brasileira de Automática, 2012. p. 4500-4507.

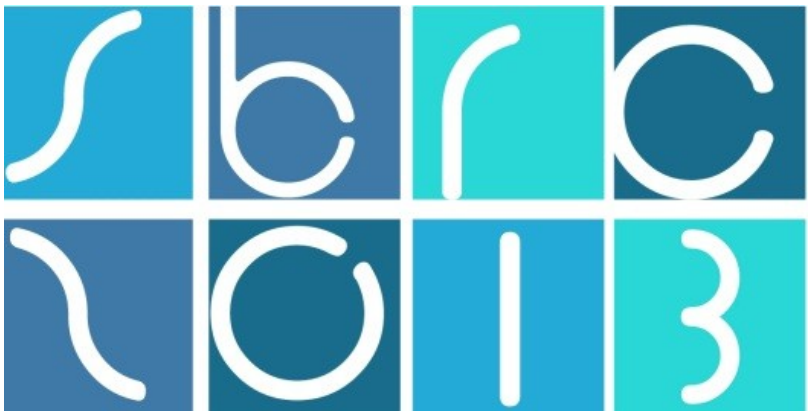
[12] Mayr, A.; Plösch, R.; Saft, M.; , "Towards an Operational Safety Standard for Software: Modelling IEC 61508 Part 3," *Engineering of Computer Based Systems (ECBS), 2011 18th IEEE International Conference and Workshops on* , vol., no., pp.97-104, 27-29 April 2011

[13] (2013) Silcore. [Online]. Available: http://www.acm.ab.ca/

[14] (2013) exSILentia. [Online]. Available: http://www.exida.com/

[15] (2013) SilSolver. [Online]. Available http://www.sis-tech.com/

[16] Weber, Taisy Silva; Cechin, Sergio Luis; Netto, Joao Cesar. Integridade de segurança em sistemas críticos de controle e instrumentação . In: *Conferência Internacional em Tecnologias Naval e Offshore : ciência e inovação* (1. : 2012 março 22-23 : Rio Grande, RS), Rio Grande, RS : FURG, 2012. [4] f.



31º Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos
Brasília-DF

XIV Workshop de Testes e Tolerância a Falhas

Sessão Técnica 2

Algoritmos Distribuídos

Replicação Máquina de Estados Dinâmica*

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

[1] [1]Departamento de Ciência da Computação
Universidade de Brasília
alchieri@cic.unb.br

[2]Faculdade de Ciências
Universidade de Lisboa
bessani@di.fc.ul.pt

[3]Departamento de Automação e Sistemas
Universidade Federal de Santa Catarina
fraga@das.ufsc.br

Resumo. *A replicação Máquina de Estados é a abordagem mais abrangente e também a mais usada na implementação de sistemas tolerantes a faltas, tanto por parada quanto bizantinas. Esta abordagem consiste em replicar os servidores e coordenar as interações entre os clientes e as réplicas dos servidores, com o intuito de que as várias réplicas apresentem a mesma evolução em seus estados. Apesar da grande maioria das concretizações de replicação Máquina de Estados terem sido desenvolvidas para ambientes estáticos, onde todas configurações do sistema como o conjunto de réplicas que o implementa nunca sofre alterações, muitas aplicações necessitam executar reconfigurações para alterar estes e outros parâmetros do sistema. Neste sentido, este artigo descreve nossos esforços para adicionar suporte a reconfigurações no sistema* BFT-SMART*, que é uma implementação de uma replicação Máquina de Estados tolerante a faltas bizantinas.*

Abstract. *State Machine Replication is an approach widely used to implement fault-tolerant systems. The idea behind this approach is to replicate the servers and to coordinate the interactions among clients and servers replicas, making all of these replicas present the same state evolution (changes). Although most of the State Machine Replication implementations were developed for static environments, where system settings (as the servers set) do not change, many applications need reconfigurations in order to change the system parameters and the servers set. In this sense, this paper describes our efforts to add support to reconfigurations in* BFT-SMART*, which is an State Machine Replication implementation able to tolerate Byzantine failures.*

1. Introdução

A replicação Máquina de Estados [Schneider 1990], também chamada de RME, é a abordagem mais abrangente e também a mais usada na implementação de sistemas tolerantes a faltas, tanto por parada [Schneider 1990] quanto bizantinas [Castro and Liskov 2002]. Esta abordagem consiste em replicar os servidores e coordenar as interações entre os clientes e as réplicas dos servidores, com o intuito de que as várias réplicas apresentem a mesma evolução em seus estados.

*Este trabalho recebeu apoio do DPP/UnB através do Edital 10/2012.

O ponto fundamental de uma replicação Máquina de Estados é a necessidade de ordenação das requisições dos clientes para serem executadas pelas réplicas (servidores) que implementam o sistema, a fim de manter o determinismo de réplicas. Para isso, é necessário o emprego de um protocolo de difusão atômica, que geralmente é implementado através de um algoritmo de consenso, onde todas as réplicas entram em acordo sobre a ordem de execução das requisições, as quais são então organizadas em uma sequência que é executada no sistema.

A grande maioria das concretizações existentes para replicação Máquina de Estados (ex.: [Castro and Liskov 2002]), bem como os protocolos propostos para difusão atômica (ex.: [Correia et al. 2006]), consideram que o conjunto de réplicas que implementam o sistema, além dos parâmetros de configuração do mesmo, nunca sofrem alterações. Uma grata exceção é o sistema SMART [Lorch et al. 2006] que implementa uma replicação Máquina de Estados onde apenas o conjunto de servidores pode ser alterado, sendo que os mesmos podem falhar apenas por parada.

A abordagem estática para replicação Máquina de Estados impossibilita, por exemplo, que durante a execução do sistema novas réplicas sejam adicionadas e/ou réplicas antigas sejam removidas e/ou que qualquer outro parâmetro de configuração (como o limite de falhas suportadas) seja alterado em tempo de execução. No entanto, muitos sistemas necessitam executar estas ações quando querem aumentar/diminuir o limite de falhas suportadas ou trocar um servidor antigo com uma configuração obsoleta por um servidor atualizado. Para isso, surge a necessidade de *reconfiguração* do sistema, tornando-o apto para a execução em ambientes dinâmicos.

Um sistema reconfigurável fornece as interfaces necessárias para sua reconfiguração, não se preocupando com aspectos relacionados com a escolha tanto do momento da reconfiguração quanto da nova configuração a ser adotada pelo sistema. De qualquer forma, os algoritmos que implementam estes sistemas devem prever a possibilidade de entradas e saídas de réplicas do mesmo, e quando consideramos uma replicação Máquina de Estados estas ações de reconfiguração devem ser sincronizadas com as execuções dos protocolos de difusão atômica do sistema, a fim de permitir que todas as réplicas executem a mesma sequência de operações (ou armazenam estados que reflitam a mesma sequência de operações), preservando assim a consistência de seus estados.

Recentemente, Lamport *et al.* [Lamport et al. 2010] apresentam uma discussão, numa visão bastante ampla e pouco aprofundada (sem apresentar protocolos e nem mesmo se aprofundar nas questões envolvidas em uma reconfiguração), sobre como reconfigurar uma replicação Máquina de Estados. Nesta abordagem, a reconfiguração da RME é realizada através da própria execução da RME, i.e., a própria RME define a nova configuração do sistema. Apesar de bastante intuitiva, existem vários problemas não triviais que devem ser tratados em uma concretização desta reconfiguração. Também em vista disto, ainda não existe nenhuma implementação de uma RME que tolera comportamento malicioso de servidores e possui capacidade de reconfiguração.

Este artigo apresenta nossos esforços para adicionar suporte a reconfigurações no sistema BFT-SMART, que é uma concretização de uma replicação Máquina de Estados tolerante a faltas bizantinas. Esta característica é importante na medida em que nestes sistemas reconfiguráveis aumenta-se significativamente a possibilidade de processos maliciosos estarem presentes no sistema, devido às computações de entradas e saídas de processos. De acordo com nossos conhecimentos, a inclusão desta funcionalidade no BFT-SMART o torna a primeira implementação de uma RME capaz de tolerar servidores maliciosos e de alterar o conjunto de

réplicas do sistema em tempo de execução.

O restante deste artigo está organizado da seguinte forma. A Seção 2 apresenta o nosso modelo de sistema. Os conceitos envolvendo uma replicação Máquina de Estados e o sistema BFT-SMART são descritos na Seção 3. A Seção 4 discute como o suporte a reconfigurações foi introduzido no BFT-SMART. A Seção 5 apresenta algumas discussões sobre os mecanismos propostos e as conclusões do trabalho são apresentadas na Seção 6.

2. Modelo de Sistema

Consideramos um sistema distribuído completamente conectado composto pelo conjunto universo de processos U, que é dividido em dois subconjuntos: um conjunto infinito de servidores $\Pi = \{s_1, s_2, ...\}$ e um conjunto infinito de clientes $C = \{c_1, c_2, ...\}$. A chegada dos processos segue o modelo de chegadas infinitas com concorrência desconhecida mas finita [Aguilera 2004]. Desta forma, em cada instante de tempo real t da execução, o número de processos executando alguma ação no sistema é desconhecido mas finito. Contudo, processos podem chegar em qualquer momento (chegadas infinitas) e também passarão a participar das computações executadas no sistema.

Os processos do sistema estão sujeitos a *faltas bizantinas* [Lamport et al. 1982], i.e., processos faltosos podem exibir qualquer comportamento, podendo parar, omitir envio ou entrega de mensagens, ou desviar de suas especificações arbitrariamente e trabalhar em conjunto com o objetivo de corromper o sistema. Um processo que apresenta comportamento de falha é dito falho (ou faltoso), de outra forma é dito correto. Como veremos a seguir, o número máximo de faltas toleradas pelo sistema, denotado por f, pode ser reconfigurado durante a execução do mesmo. No entanto, sempre é necessário pelo menos $3f + 1$ réplicas para tolerar até f réplicas faltosas no sistema em um dado momento.

Com relação ao modelo de sincronia, consideramos um sistema parcialmente síncrono [Dwork et al. 1988]. A ideia por trás destes modelos é de que o sistema trabalha de forma assíncrona (não respeitando nenhum limite de tempo) a maior parte do tempo. Porém, durante períodos de estabilidade, o tempo para transmissão de mensagens é limitado. Além disso, as comunicações entre os processos são realizadas através de canais ponto-a-ponto confiáveis e autenticados.

Finalmente, cada processo possui um par distinto de chaves (chave pública e privada) para usar um sistema de criptografia assimétrica. Cada chave privada é conhecida apenas pelo seu próprio dono, por outro lado todos os processos conhecem todas as chaves públicas. Cada processo do sistema (cliente ou servidor) possui um identificador único, representado por este par de chaves obtidas junto a uma autoridade certificadora, sendo inviável a obtenção de identificadores adicionais por processos faltosos com o objetivo de lançar um ataque *Sybil* [Douceur 2002] contra o sistema.

3. Replicação Máquina de Estados

A replicação Máquina de Estados [Schneider 1990] é a abordagem mais abrangente e também a mais usada na implementação de sistemas tolerantes a faltas, tanto por parada [Schneider 1990] quanto bizantinas [Castro and Liskov 2002]. Esta abordagem consiste em replicar os servidores e coordenar as interações entre os clientes e as réplicas dos servidores, com o intuito de que as várias réplicas apresentem a mesma evolução em seus estados, i.e., em qualquer ponto da execução do sistema distribuído todas as réplicas devem possuir o mesmo estado.

Para isso, a replicação Máquina de Estados exige que todas as réplicas, *(i)* partindo de

um mesmo estado e *(ii)* executando o mesmo conjunto de requisições na mesma ordem, *(iii)* cheguem ao mesmo estado final, o que define o determinismo de réplicas.

O item *(i)* é facilmente garantido, bastando iniciar todas as réplicas com o mesmo estado (i.e., iniciar todas as variáveis que representam o estado com os mesmos valores nas diversas réplicas). Para prover o item *(ii)*, é necessária a utilização de um protocolo de difusão atômica como mostra a Figura 1. O problema da *difusão atômica* [Hadzilacos and Toueg 1994], também conhecido como difusão com ordem total, consiste em fazer com que todos os processos corretos, membros de um grupo, entreguem todas as mensagens difundidas neste grupo na mesma ordem. A Figura 1 mostra um exemplo onde dois clientes estão concorrentemente acessando quatro servidores. Neste caso, através de um protocolo de difusão atômica, suas requisições são entregues e executadas na mesma ordem pelos servidores, i.e., caso um servidor execute primeiro a requisição $op1$ (requisição do cliente 1) de depois a requisição $op2$ (requisição do cliente 2), então todos os outros servidores também executarão primeiramente $op1$ e depois $op2$, mantendo a ordem de entrega/execução.

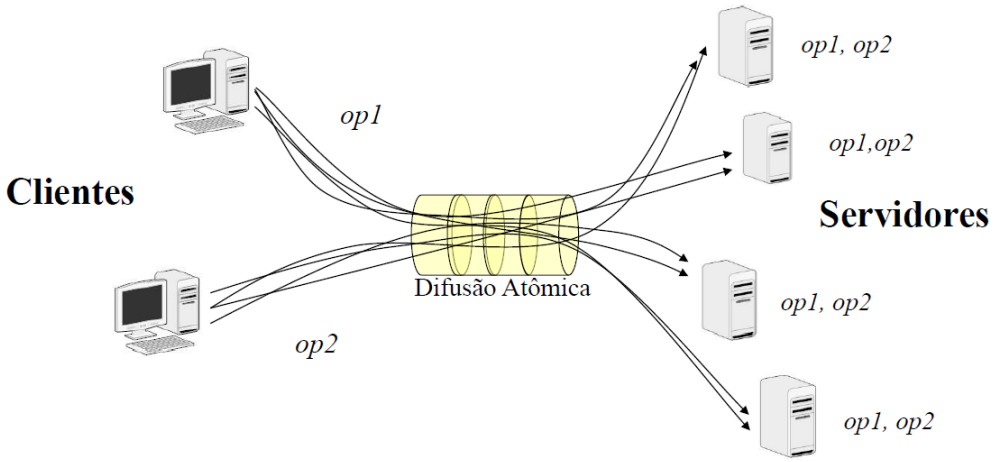


Figura 1. Replicação Máquina de Estados.

Finalmente, para alcançar o item *(iii)*, é necessário que as operações executadas pelas réplicas sejam deterministas, i.e., que a execução de uma mesma operação (com os mesmos parâmetros) resulte no mesmo resultado nas diversas réplicas. Além disso, o estado produzido (a mudança no estado) por esta operação deve ser o mesmo nas várias réplicas do sistema.

Um resultado teórico interessante é que a difusão atômica e o consenso são problemas equivalentes em sistemas distribuídos onde os processos então sujeitos tanto a faltas de parada [Chandra and Toueg 1996] quanto a faltas bizantinas [Correia et al. 2006]. O *problema do consenso* [Hadzilacos and Toueg 1994] consiste em fazer com que todos os processos corretos acabem por decidir o mesmo valor, o qual deve ter sido previamente proposto por algum dos processos do sistema. Por exemplo, para implementar difusão atômica através de um protocolo de consenso, basta que os processos utilizem este protocolo para entrarem em acordo (propriedade fundamental do consenso [Hadzilacos and Toueg 1994]) acerca da ordem de entrega das mensagens (requisições).

3.1. BFT-SMaRt: Implementação de Replicação Máquina de Estados

O BFT-SMART [Bessani et al. 2011] representa a concretização de uma replicação Máquina de Estados [Schneider 1990] tolerante a faltas bizantinas [Lamport et al. 1982]. Esta biblioteca

de replicação foi desenvolvida na linguagem de programação Java e implementa um protocolo similar aos outros protocolos para tolerância a faltas bizantinas (ex.: [Castro and Liskov 2002]), mas que se preocupa tanto com o desempenho do sistema quanto com a corretude do mesmo nos mais diversos cenários originados pelo comportamento malicioso de réplicas.

O BFT-SMART surgiu da camada de replicação do sistema DEPSPACE [Bessani et al. 2008], o qual representa a implementação de um espaço de tuplas tolerante a faltas bizantinas que utiliza replicação Máquina de Estados para garantir a consistência dos estados das réplicas do sistema. Atualmente, o BFT-SMART conta com protocolos para *checkpoints* e transferência de estados, tornando-se assim uma biblioteca completa para RME, a qual foi desenvolvida seguindo os seguintes princípios:

- Java: a escolha desta linguagem de programação visa a obtenção de uma implementação que apresenta características de portabilidade, segurança, facilidade de programação e manutenção.
- Modularidade: o BFT-SMART foi projetado de forma modular, apresentando uma notável separação entre os protocolos de consenso (o algoritmo de consenso utilizado é o *Paxos at War* [Zielinski 2004]), de difusão atômica, de *checkpoints* e de transferência de estado.
- Desprovido de otimizações que aumentam a complexidade dos algoritmos: além de adicionar complexidade, a utilização de otimizações "frágeis" torna os protocolos mais susceptíveis a ataques de degradação de performance [Clement et al. 2009]. Desta forma, o BFT-SMART não implementa algumas otimizações como a execução do acordo sobre *hashes* e especulação.

Estes princípios de projeto fazem do BFT-SMART a concretização de uma biblioteca de replicação razoavelmente estável e completa, que pode ser usada tanto em pesquisas quanto no desenvolvimento de protótipos.

3.1.1. Arquitetura Básica do BFT-SMART

Esta seção discute os aspectos principais da arquitetura do BFT-SMART, os quais são importantes para entender a forma de como as reconfigurações foram incorporadas neste sistema. Primeiramente, vamos analisar como é a arquitetura de cada réplica do sistema (Figura 2). Os clientes acessam as réplicas através de um conjunto de *threads* (uma *thread* é iniciada para cada cliente), sendo que as requisições de cada cliente são adicionadas em uma fila separada. Sempre que existirem requisições para serem executadas, uma *thread* (*proposer*) iniciará uma instância do consenso para definir uma ordem de entrega da mesma (como veremos adiante, na verdade cada instância do consenso define a ordem de execução de várias requisições). Durante este processo, a réplica se comunica com as outras através de um outro conjunto de *threads*, sendo que cada uma destas *threads* é responsável pela conexão com uma das outras réplicas. Além disso, existe um conjunto de *threads* (também uma para cada réplica) responsável por receber as mensagens enviadas pelas outras réplicas. Todo o processamento necessário para garantir a autenticidade destas mensagens é executado nestas *threads*. A utilização destes conjuntos de *threads* faz com que o BFT-SMART apresente um bom desempenho em CPUs multicore.

Finalmente, quando a ordem de execução de uma requisição é definida, a mesma é adicionada em uma fila para então ser entregue à aplicação (*ServiceReplica*) por uma outra *thread*. Após o processamento da requisição, uma resposta é enviada ao cliente que solicitou tal requisição. O cliente, por sua vez, determina que uma resposta para sua requisição é válida

assim que o mesmo receber pelo mesmo $f + 1$ respostas iguais, garantindo que pelo menos uma réplica correta obteve tal resposta.

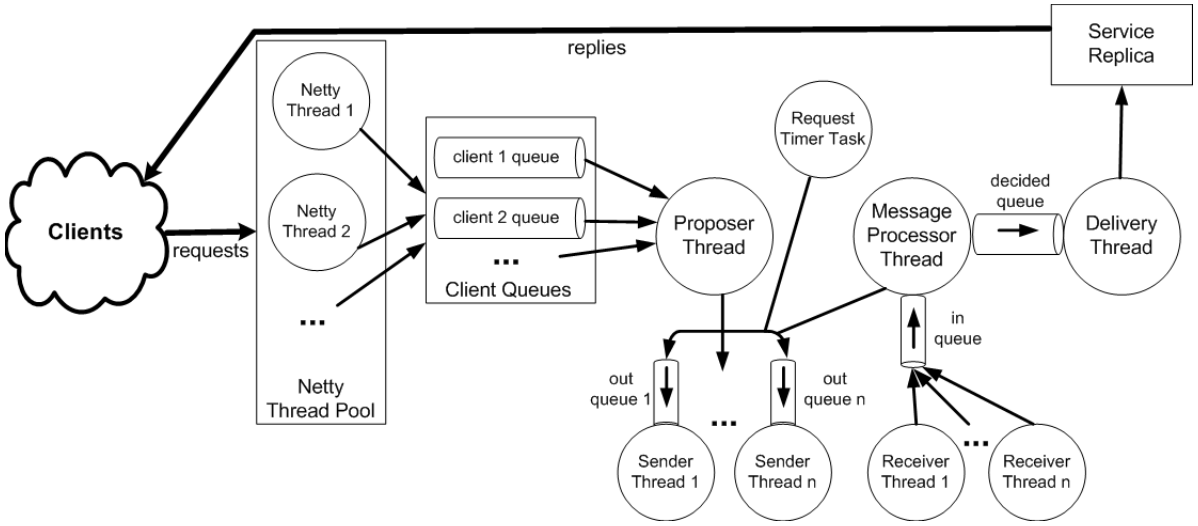


Figura 2. Arquitetura de uma Réplica do BFT-SMaRt.

Como já comentado, o protocolo de ordenação das requisições utiliza o algoritmo de consenso *Paxos at war* [Zielinski 2004]. Basicamente, este protocolo funciona da seguinte forma: o valor de decisão da instância i do *Paxos* é a i-ésima requisição a ser entregue para a aplicação. Desta forma, a entrega das requisições segue a mesma ordem nas diversas réplicas. Apesar da aparente simplicidade, alguns outros problemas devem ser tratados: *(1)* no protocolo de consenso, a réplica líder pode propor qualquer valor de tal forma que a decisão não seja uma requisição válida e autêntica (requisições forjadas); e *(2)* um líder malicioso pode executar o protocolo de consenso corretamente, mas nunca propor uma ordem para requisições de determinado(s) cliente(s) (negação de serviço para clientes). No BFT-SMaRt, os seguintes mecanismos foram incorporados ao sistema para evitar que réplicas maliciosas sejam capazes de executar qualquer uma destas ações:

- Autenticidade de Requisições: a autenticidade das requisições é garantida por meio de assinaturas digitais, i.e., os clientes devem assinar suas requisições. Desta forma, qualquer réplica é capaz de verificar a autenticidade das requisições e uma proposta para ordenação, a qual contém a requisição a ser ordenada, somente é aceita por uma réplica correta após a autenticidade desta requisição ser verificada.
- Garantia de Ordenação de Requisições: caso uma requisição não seja ordenada dentro de um determinado tempo, o sistema força a troca da réplica líder. Para cada requisição r recebida em determinada réplica i, um tempo limite para ordenação é associado à r. Caso este tempo se esgotar, i envia r para todas as réplicas e define um novo tempo para sua ordenação. Isto garante que todas as réplicas recebem r, pois um cliente malicioso pode ter enviado r apenas para alguma(s) réplica(s), tentando forçar uma troca de líder. Caso este tempo se esgotar novamente, i solicita a troca de líder, que apenas é executada após $f + 1$ réplicas solicitarem esta mudança, impedindo que uma réplica maliciosa force trocas de líder.

A escolha do método de reconfiguração do sistema está diretamente relacionada com a forma de como o protocolo de ordenação das requisições é implementado no mesmo [Lamport et al. 2010]. Neste sentido, a seguir descrevemos duas características fundamentais do protocolo de ordenação do BFT-SMaRt:

- Ordenação em lote: no BFT-SMaRt, cada instância do consenso define a ordem de entrega de um lote de requisições ao invés de apenas uma única requisição. As requisições de um lote devem ser entregues seguindo uma mesma ordem em todos as réplicas (ex.: entregues de acordo com a ordem de seus identificadores). Esta abordagem aumenta o desempenho do sistema, uma vez que várias requisições são entregues através da execução de uma única instância do consenso. No entanto, é possível que requisições de reconfiguração do sistema (que são ordenadas juntamente com as requisições de clientes – Seção 4.1) estejam localizadas no meio de determinado lote, "misturadas" com requisições de clientes, e isto deve ser previsto pelos protocolos de reconfiguração, como veremos na Seção 4.1.
- Eliminação da concorrência na execução de instâncias do consenso: outra abordagem que visa aumentar o desempenho do sistema é a execução em paralelo de várias instâncias do algoritmo de consenso, de modo que várias requisições (ou lotes de requisições) sejam ordenadas simultaneamente. Além desta abordagem trazer mais complexidade aos protocolos, a mesma possibilita que líderes faltosos degradem o desempenho do sistema, uma vez que é necessário executar instâncias do consenso (iniciadas por estes processos maliciosos) mesmo quando as propostas são para definir a ordem de entrega de requisições forjadas. Neste caso, a instância do consenso define a ordem para uma requisição *nop* (*no operation*), o que é necessário para não "travar" a entrega das mensagens [Castro and Liskov 2002]. Em vista disso, o BFT-SMaRt opta por executar uma única instância do consenso por vez e preserva o desempenho do sistema através da ordenação em lotes [Bessani et al. 2008]. Como descrito por Lamport *et al.* [Lamport et al. 2010], esta abordagem também torna o protocolo de reconfiguração menos complexo (Seção 4.1) quando comparado com soluções onde várias instâncias do consenso são executadas em paralelo.

3.1.2. Usando o BFT-SMaRt

A forma de utilização do BFT-SMaRt, para programação de uma aplicação tolerante a faltas bizantinas através de replicação Máquina de Estados, é bastante simples. A Figura 3 apresenta a API para clientes e servidores, mostrando a classe que deve ser instanciada pelo clientes para acessar o sistema, bem como a classe que deve ser estendida pelos servidores para implementar o serviço replicado.

Para acessar o serviço replicado, um cliente do BFT-SMaRt apenas deve instanciar uma classe *ServiceProxy* com um arquivo de configuração contendo o endereço (IP e porta) de cada um dos servidores, bem como suas chaves públicas. Então, sempre que o cliente desejar enviar alguma requisição para as réplicas (servidores), o mesmo deve invocar o método *invoke* especificando a requisição (serializada em um *array* de *bytes*) e indicando se tal requisição é apenas de leitura. Requisições de apenas leitura não modificam o estado das réplicas e, deste modo, não precisam ser ordenadas, sendo entregues diretamente para a aplicação.

Por outro lado, para implementar o servidor, cada réplica deve estender a classe *ServiceReplica* e implementar os métodos abstratos que são invocados quando uma requisição deve ser executada (é entregue pelo protocolo de ordenação) ou quando é necessário obter/atualizar o estado da réplica. Os métodos para atualização ou obtenção dos estados das réplicas são utilizados pelos mecanismos de *checkpoints* e transferência de estados, sendo indispensáveis para a recuperação de réplicas faltosas ou atualização de réplicas atrasadas [Bessani et al. 2011].

Note que o método *executeCommand* também fornece o identificador do cliente, um

timestamp e um conjunto de *nonces* randômicos, definidos pela réplica líder da instância do consenso que definiu a ordem de execução da requisição correspondente. Como é garantido que todas as réplicas recebem a requisição com o mesmo *timestamp* e conjunto de *nonces*, é possível implementar ações tipicamente não deterministas, como leitura do valor do *clock* ou geração de números randômicos, de forma determinista nas réplicas.

```
//API do Cliente
public class ServiceProxy ... {
  ...
  public byte[] invoke(byte[] command, boolean readOnly);
  ...
}


//API do Servidor
public abstract class ServiceReplica ...{
   ...
   public abstract byte[] executeCommand(int clientId, long timestamp,
                                                 byte[] nonces, byte[] command);
   public abstract byte[] serializeState();
   public abstract byte[] deserializeState(byte[] state);

}
```

Figura 3. API do BFT-SMaRt para clientes e servidores.

A Figura 4 apresenta as interações que ocorrem no sistema para a execução de uma requisição que deve ser ordenada, i.e., uma requisição que altera o estado das réplicas. Primeiramente, o cliente envia a requisição assinada para todas as réplicas do sistema, através da classe *ServiceProxy* (método *invoke*). Após receber a requisição, as réplicas executam o protocolo de ordenação para definir uma ordem de entrega para a mesma (passo 2).

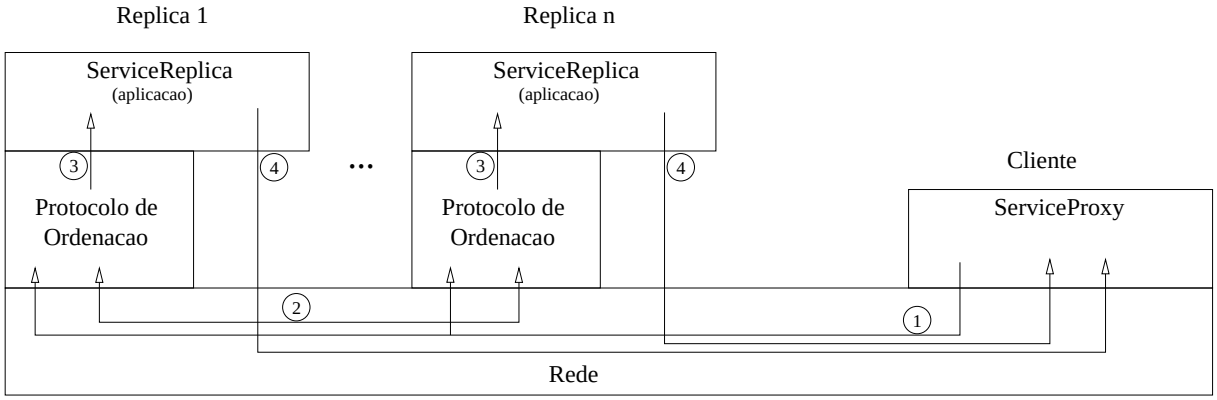


Figura 4. Interações no BFT-SMaRt.

Quando as requisições anteriores já tiverem sido entregues em determinada réplica, a mesma executa o método *executeCommand* para entregar esta requisição para a aplicação (*ServiceReplica* - passo 3). Após executar a requisição, cada réplica envia a resposta para o cliente (passo 4), que recebe estas mensagens na classe *ServiceProxy*. Quando pelo menos $f + 1$ respostas iguais são recebidas, o protocolo termina e esta resposta é o resultado da execução do método *invoke*.

4. Reconfiguração de uma Replicação Máquina de Estados

Diversas maneiras de reconfigurar o conjunto de réplicas que implementam uma Máquina de Estados são discutidas por Lamport *et al.* [Lamport et al. 2010]. Nestas abordagens, a própria

Máquina de Estados é usada na definição da nova configuração do sistema, fazendo com que todas as réplicas concordem com a nova configuração (visão) de forma semelhante aos protocolos de *group membership* [Chockler et al. 2001].

Seguindo esta abordagem, as reconfigurações do sistema ocorrem entre execuções do protocolo de ordenação (consenso), de modo que a própria RME é usada para definir a nova configuração do sistema. De fato, não faz muito sentido reconfigurar o sistema em meio a uma execução do protocolo de consenso, visto que estes protocolos geralmente terminam rapidamente (em alguns milissegundos – [Castro and Liskov 2002, Bessani et al. 2008]) e a RME em execução já fornece um suporte bastante forte que pode ser utilizado nas reconfigurações. Reconfigurar o sistema em meio a uma execução do consenso implicaria em adicionar muita complexidade neste protocolo, além de atrelar o sistema a um protocolo de consenso específico, i.e., com capacidade de reconfiguração. Além disso, estas limitações seriam introduzidas no sistema sem ganhos práticos, pois a reconfiguração do mesmo exigiria os seguintes passos:

1. O consenso em execução deveria ser paralizado;
2. O sistema deveria ser reconfigurado, onde a nova configuração possivelmente seria escolhida através de um consenso;
3. Então, o consenso anteriormente paralizado poderia ser finalizado.

Como veremos na seção seguinte, a reconfiguração através da própria RME demanda no máximo uma execução do consenso (ou uma execução do protocolo de ordenação da RME) para definir a nova configuração do sistema. Além disso, pode ser necessário que algum cliente execute novamente sua requisição utilizando a visão (configuração) mais atual do sistema, para evitar o acesso a réplicas obsoletas que já não fazem mais parte do sistema.

4.1. Reconfigurando o BFT-SMaRt

A reconfiguração do BFT-SMART segue a ideia de utilizar a própria RME para obter a configuração atualizanda, sendo bastante simples. Em termos gerais, o protocolo funciona da seguinte forma:

1. Primeiramente, algum processo envia uma requisição de reconfiguração do sistema, de forma idêntica a um cliente que acessa o sistema normalmente para a execução de alguma operação pela Máquina de Estados.
2. Esta requisição é ordenada junto com as requisições dos clientes, i.e., a mesma é tratada como uma requisição normal pelo protocolo de ordenação, que definirá a ordem de entrega para um lote de requisições que conterá tal requisição de reconfiguração.
3. Quando este lote de requisições for entregue em determinada réplica, a mesma executa todas as operações normais (requisições executadas pela aplicação) para então executar todas as operações de reconfiguração contidas neste lote. Note que um lote pode conter mais de um pedido de reconfiguração e todas as alterações no sistema, definidas pelas requisições deste lote, são processadas de uma só vez após a execução das requisições normais deste lote.
4. Uma nova configuração do sistema é definida e enviada para os clientes, bem como para as réplicas que estão entrando no sistema.
5. As réplicas que estão entrando no sistema atualizam seus estados a partir do estado das réplicas da configuração antiga, as quais fornecem seus estados obtidos até a execução da reconfiguração.
6. Por fim, todas as réplicas da nova configuração iniciam a execução da Máquina de Estados com o estado atual do sistema.

Como no BFT-SMART instâncias de consenso não são executadas em paralelo, não é necessário se preocupar com instâncias utilizando a configuração antiga, uma vez que a instância de consenso seguinte, que definirá a ordem de entrega para o lote de requisições seguinte, já é inicializada utilizando-se a nova configuração do sistema.

Desta forma, o BFT-SMART com capacidade de reconfiguração podem ser entendido como uma sequência de Máquinas de Estados, onde a máquina anterior na sequência escolhe a configuração da máquina seguinte e é finalizada (parada). Então, a máquina seguinte é inicializada com esta configuração a partir do estado final da máquina anterior, e assim por diante.

Existem duas formas de reconfigurações suportadas pelo BFT-SMART: *(1)* no primeiro tipo de reconfiguração a própria réplica (servidor) solicita sua entrada ou saída do sistema; e no segundo tipo *(2)*, mais abrangente, uma terceira parte confiável (TTP - *trusted third party*) tem o poder de reconfigurar o sistema, requisitando entradas e saídas de réplicas e/ou solicitando alterações nos parâmetros de configuração do sistema, como por exemplo o número máximo de faltas toleradas.

4.1.1. Visões

Antes de discutir as formas de reconfiguração do BFT-SMART, vamos definir os componentes que formam as visões (configurações) do sistema, uma vez que este conceito é fundamental em sistemas renconfiguráveis, pois sempre que uma reconfiguração ocorre, uma nova visão mais atual é gerada para o mesmo. Esta nova visão reflete os pedidos de alterações que motivaram a execução da reconfiguração.

Sendo assim, cada visão v do sistema contém um identificador único ($v.id$), que nada mais é do que um contador que é incrementado na medida que reconfigurações são executadas. Desta forma, através de seus identificadores, podemos definir facilmente qual de duas visões é mais atual no sistema.

Além disso, cada visão contém os identificadores das réplicas (servidores) que fazem parte de tal visão, bem como o valor dos parâmetros reconfiguráveis no sistema. Atualmente, o único parâmetro reconfigurável no BFT-SMART, além do conjunto de servidores, é o limite f de falhas suportadas pelo sistema. Para cada visão v, o número de réplicas presentes nesta visão $v.n$ deve ser $v.n \geq 3v.f + 1$, onde $v.f$ representa o número de falhas de réplicas de v suportadas pelo sistema.

4.1.2. Reconfiguração a partir da própria Réplica

Este tipo de reconfiguração é mais restrita, onde uma determinada réplica (servidor) apenas é capaz de solicitar sua própria entrada e/ou saída do sistema. Deste modo, nenhum outro parâmetro do sistema pode ser alterado. A Figura 5 apresenta os métodos adicionados na API dos servidores (classe *ServiceReplica*), os quais permitem a solicitação de suas entradas (*join*) e/ou saídas (*leave*).

Na execução de qualquer um destes métodos, o servidor acessa o sistema como se fosse um cliente e envia a sua solicitação de reconfiguração, que é processada como descrito anteriormente. Após ordenada, esta requisição não é entregue para a aplicação, mas para um módulo de reconfiguração (*ReconfigurationManager*) onde uma nova configuração (visão) é gerada para o sistema. Por fim, uma resposta é enviada ao solicitante para informar a execução de sua

requisição de reconfiguração.

```
//API do Servidor
public abstract class ServiceReplica ...{
   ...
   public void join();
   public void leave();
}
```

Figura 5. Métodos adicionais para reconfiguração do BFT-SMART.

A Figura 6 apresenta as interações neste procedimento, onde os passos $3b$ e $4b$ representam a execução de uma requisição de reconfiguração, enquanto os passos $3a$ e $4a$ referem-se à execução de uma requisição normal. Através do protocolo de ordenação, estes passos são sincronizados em todas as réplicas presentes em determinada visão, i.e., as requisições são entregues para a aplicação ou para o módulo de reconfiguração de forma ordenada, seguindo a mesma sequência em todas as réplicas presentes em determinada visão.

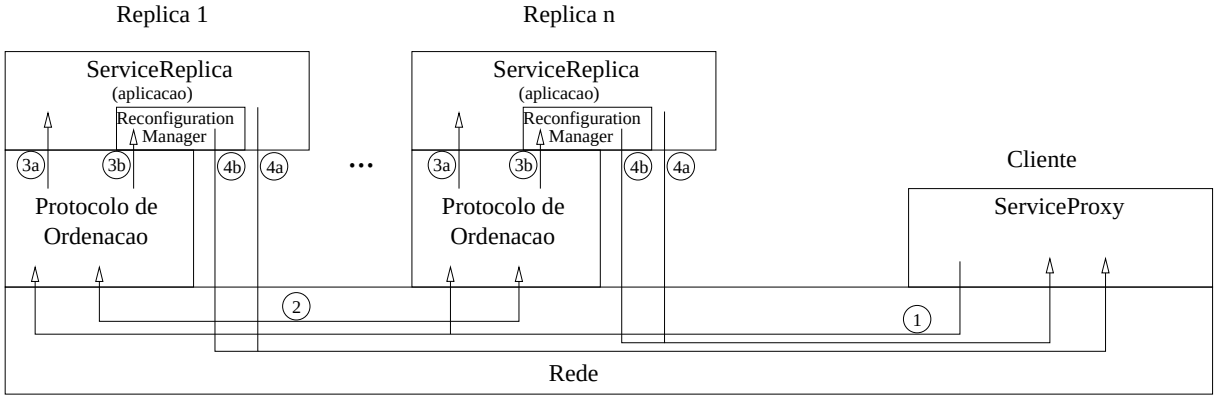


Figura 6. Interações no BFT-SMART reconfigurável.

No caso de uma solicitação para entrada no sistema, a requisição contém o identificador e o endereço (IP e porta) do servidor solicitante, de modo que todos os servidores do sistema passam a conhecer este novo servidor. Já a resposta obtida desta solicitação contém a visão na qual o servidor entrou no sistema (nova visão gerada pelo *ReconfigurationManager*) e todos os demais dados necessários para a inicialização da réplica (estado e parâmetros de configuração do BFT-SMART). Além disso, as conexões são atualizadas em todos os servidores do sistema, de acordo com a nova visão do mesmo. Por fim, todos os servidores da nova visão passam a participar do sistema. Todo este processamento é completamente transparente no BFT-SMART, sendo que apenas deve-se executar o método *join* (Figura 5) para que determinada réplica entre no sistema. A computação de um *leave* é mais simples, pois não é necessário atualizar o estado de uma réplica que está deixando o sistema.

4.1.3. Reconfiguração a partir da TTP

Este tipo de reconfiguração é mais abrangente, onde uma terceira parte confiável (TTP - *trusted third party*) é capaz de modificar tanto o grupo de réplicas que implementam a Máquina de Estados quanto as configurações do sistema, como por exemplo o limite f de faltas suportadas. A Figura 7 apresenta a API da TTP, onde podemos verificar quais são os métodos que devem ser usados para reconfigurações do sistema.

A TTP deve ser criada da mesma forma de um cliente normal (Seção 3.1.2). Posteriormente, para adicionar ou remover servidores, devemos utilizar os métodos *addServer* e *removeServer*, respectivamente. Para cada novo servidor, é necessário informar seu identificador e seu endereço (IP e porta). Também é possível modificar o limite f de faltas suportadas pelo sistema através do método *setF*, onde esta alteração apenas é executada caso o número de réplicas na nova visão v continue obedecendo ao limite $v.n \geq 3v.f + 1$.

```
//API da TTP
public class TTP{
   ...
   public void addServer(int id, String ip, int port);
   public void removeServer(int id);
   public void setF(int f);
   public void executeUpdates();
   public void close();
}
```

Figura 7. API da TTP.

Com o intuito de evitar que para cada alteração desejada seja necessário que o sistema execute uma nova reconfiguração, a TTP armazena todas as alterações solicitadas através destes métodos (*addServer*, *removeServer* e *setF*) e envia apenas uma única requisição de reconfiguração contendo todas estas alterações. Este processamento é executado através do método *executeUpdates*, onde a TTP acessa o sistema como um cliente (assim como descrito na seção anterior) para requisitar a reconfiguração do mesmo.

Quando a resposta é recebida na TTP, a mesma envia uma mensagem para os servidores que estão entrando no sistema, informando-os sobre a visão na qual tais servidores entraram no sistema e sobre os demais dados necessários para a inicialização destas réplicas (estado e parâmetros de configuração do BFT-SMART). Após isso, as réplicas atualizam suas conexões, de acordo com a nova visão do sistema. Por fim, todos os servidores da nova visão passam a participar do sistema. Todo este processamento é completamente transparente no BFT-SMART, sendo que apenas deve-se executar o método *executeUpdates* (Figura 7) para que as atualizações solicitadas sejam refletidas no sistema.

Após a execução do método *executeUpdates*, a TTP fica pronta para ser usada novamente em uma outra reconfiguração, onde novas alterações serão executadas no sistema. No entanto, é possível finalizar a TTP através do método *close* que fecha todas as conexões da TTP com os servidores do sistema. De qualquer forma, caso seja necessário, posteriormente é possível criar uma outra TTP para executar uma nova reconfiguração.

5. Discussão

Esta seção apresenta algumas discussões sobre aspectos relacionados com os protocolos de reconfiguração do BFT-SMART. Note que os problemas discutidos e as soluções propostas e adotadas em nosso sistema são válidas para qualquer sistema reconfigurável.

5.1. Lidando com Reconfigurações

Para evitar o acesso a dados obsoletos, os clientes do BFT-SMART sempre devem acessar a configuração (visão) mais atual do sistema. Desta forma, um cliente c anexa em suas requisições o identificador da sua visão corrente e os servidores verificam (imediatamente antes da execução e após a ordenação da requisição – passo $3a$ ou $3b$ da Figura 6) se tal cliente está utilizando a visão mais atual do sistema. Se este for o caso, a requisição é executada normalmente. Caso

contrário, os servidores enviam uma resposta para c contendo a nova visão, e então, c envia novamente sua requisição considerando esta nova visão do sistema. Este processamento é completamente transparente no BFT-SMART, sendo apenas necessário invocar as operações normalmente no *ServiceProxy* (Seção 3.1.2).

Note que, considerando v a visão atual do sistema, para uma requisição ser ordenada e entregue, a mesma deve atingir pelo menos um servidor correto de v, onde a autenticidade desta requisição é comprovada pela assinatura do cliente. Desta forma, um cliente que utiliza uma visão antiga w apenas terá sua requisição ordenada e obterá uma resposta contendo a visão atualizada v, caso $w \cap v$ contenha pelo menos um servidor correto. Para relaxar esta necessidade, as visões do sistema deveriam ser armazenadas também em algum lugar padrão, a partir do qual os clientes poderiam obtê-las, como discutido na seção seguinte.

5.2. Armazenamento das Visões

Atualmente, as visões do BFT-SMART são armazenadas apenas pelos próprios servidores que implementam o sistema. Desta forma, os clientes apenas atualizam suas visões nos casos onde suas requisições chegam em algum servidor correto da visão atual do sistema, como discutido na seção anterior.

No entanto, é factível que clientes permaneçam um longo período de tempo sem acessar o sistema, de forma que sua visão não contenha nenhum servidor da visão atual. Nestes casos, tais clientes não conseguem atualizar suas visões. Este mesmo problema ocorre com clientes que desejam acessar o sistema somente após um conjunto de reconfigurações levar o sistema para uma visão que não contém nenhum servidor da visão inicial. Note que este problema afeta apenas os clientes, uma vez que os servidores são sempre informados sobre as novas visões.

Para resolver este problema, é necessário que as visões sejam armazenadas em algum lugar padrão, a partir do qual qualquer cliente possa obtê-las. Neste sentido, é necessário que os servidores emitam certificados (conjunto de assinaturas) que atestem a autenticidade de uma visão. Para aumentar o grau de tolerância a faltas, as visões podem ser armazenadas em vários lugares ou em algum serviço tolerante a faltas bizantinas. Nesta abordagem, sempre que um cliente não tenha sua requisição atendida dentro de um determinado tempo, o mesmo deve verificar se está utilizando a visão atual do sistema.

6. Conclusões

Este artigo apresentou os nossos esforços na concretização de uma replicação Máquina de Estados dinâmica, onde o conjunto de réplicas e o limite f de faltas suportadas pelo sistema podem sofrer reconfigurações em tempo de execução. Além disso, os mecanismos propostos para reconfiguração também fornecerão suporte para que outros parâmetros de configuração do sistema possam ser alterados durante a execução do mesmo.

Seguimos uma das abordagens para reconfiguração de uma replicação Máquina de Estados apresentada por Lamport *et al.* [Lamport et al. 2010], onde adicionamos mecanismos que implementam esta abordagem no BFT-SMART. Além da reconfiguração propriamente dita, vários outros aspectos que fogem do escopo deste artigo foram considerados em nossa implementação, como a abertura e o término de conexões de forma segura (evitando que múltiplas conexões sejam estabelecidas para determinada réplica), refletindo a visão atual do sistema. Todos os códigos gerados estão disponíveis na página do BFT-SMART [Bessani et al. 2011].

Referências

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

Bessani, A. N., Alchieri, E. A. P., Correia, M., and da Silva Fraga, J. (2008). Depspace: a byzantine fault-tolerant coordination service. *SIGOPS Oper. Syst. Rev.*, 42(4):163–176.

Bessani, A. N., Alchieri, E. A. P., and Souza, P. (2011). Bft-smart: High-performance byzantine-fault-tolerant state machine replication. http://code.google.com/p/bft-smart/.

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

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

Chockler, G. V., Keidar, I., and Vitenberg, R. (2001). Group communication specifications: A comprehesive study. *ACM Computing Surveys*, 33(4):427–469.

Clement, A., Wong, E., Alvisi, L., Dahlin, M., and Marchetti, M. (2009). Making byzantine fault tolerant systems tolerate byzantine faults. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 153–168. USENIX Association.

Correia, M., Neves, N. F., and Veríssimo, P. (2006). From consensus to atomic broadcast: Time-free Byzantine-resistant protocols without signatures. *The Computer Journal*, 49(1).

Douceur, J. (2002). The sybil attack. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems*.

Dwork, C., Lynch, N. A., and Stockmeyer, L. (1988). Consensus in the presence of partial synchrony. *Journal of ACM*, 35(2):288–322.

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

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

Lamport, L., Malkhi, D., and Zhou, L. (2010). Reconfiguring a state machine. *SIGACT News*, 41:63–73.

Lorch, J. R., Adya, A., Bolosky, W. J., Chaiken, R., Douceur, J. R., and Howell, J. (2006). The smart way to migrate replicated stateful services. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 103–115, New York, NY, USA. ACM.

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

Zielinski, P. (2004). Paxos at War. Technical Report UCAM-CL-TR-593, University of Cambridge Computer Laboratory, Cambridge, UK.

Algoritmo de Consenso Genérico em Memória Compartilhada

Cátia Khouri[1], **Fabíola Greve**[1]

[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. *Consensus is a fundamental problem for the development of reliable distributed systems. However, in asynchronous environments prone to failures, it is necessary to extend the system with some mechanism that provides the minimum synchrony necessary to circumvent the impossibility of consensus. In this paper, we present a generic consensus algorithm for asynchronous system with shared memory that can be instantiated with a failure detector $\diamondsuit\mathcal{S}$ or Ω. The algorithm is optimal regarding the number of registers it uses and it tolerates $(n-1)$ failures. This solution for shared memory favors the use of consensus in modern applications developed, for example, on multicore architectures and Storage Area Networks (SAN).*

Resumo. *O consenso é um problema fundamental para o desenvolvimento de sistemas distribuídos confiáveis. Porém, em ambientes assíncronos sujeitos a falhas, é preciso estender o sistema com algum mecanismo que forneça o sincronismo mínimo necessário para contornar a impossibilidade do consenso. Neste artigo, apresentamos um algoritmo de consenso genérico para um sistema assíncrono com memória compartilhada que pode ser instanciado com um detector de falhas $\diamondsuit\mathcal{S}$ ou Ω. O algoritmo é ótimo quanto ao número de registradores que utiliza e tolera $(n-1)$ falhas. Essa solução para memória compartilhada favorece o uso do consenso em aplicações modernas desenvolvidas, por exemplo, sobre arquiteturas multicore e Storage Area Networks (SAN).*

1. Introdução

Sistemas distribuídos tolerantes a falhas devem continuar a prover serviços a despeito de falhas em seus nós ou canais de comunicação. É fundamental que mesmo com a falha de alguns participantes, os processos que operam corretamente concordem com relação a determinada informação, para manter a integridade do sistema. É o caso, por exemplo, de sistemas de bancos de dados onde vários processos devem decidir se uma transação deve ser efetivada ou cancelada. Geralmente, quando um processo executa sua computação local com sucesso, posiciona-se favorável à efetivação; caso contrário, manifesta-se pelo cancelamento. Os vários processos então coordenam suas ações para chegar a um *acordo*.

Dentre os problemas de acordo, o *consenso* [Chandra and Toueg 1996, Lo and Hadzilacos 1994] é o mais importante. Ele pode ser visto como uma arcabouço geral de acordo e a maneira mais natural de encapsular esse problema. Informalmente,

o problema do consenso diz respeito a um conjunto de processos que devem concordar sobre um valor (ou conjunto de valores). O problema do consenso está no coração de protocolos como os de sincronização, difusão, reconfiguração, replicação de dados, leitura de sensores e outros. Entretanto, é bem sabido que não existe solução determinística para este problema em sistemas puramente assíncronos sujeitos a falhas, nos modelos de passagem de mensagens e de memória compartilhada [Fischer et al. 1985, Loui and Abu-Amara 1987]. Esse resultado tem motivado o surgimento de abordagens alternativas, enriquecendo-se o sistema com suposições adicionais de sincronia.

Considerando que essa impossibilidade se dá pela dificuldade em se distinguir, num sistema assíncrono, se um processo falhou ou se está muito lento, [Chandra and Toueg 1996] propõem estender o sistema com *detectores de falhas não confiáveis*. Estes se constituem de módulos distribuídos, cada um associado a um processo, que dão dicas sobre processos falhos no sistema, as quais podem ser corretas ou não. Independente disso, algoritmos de consenso corretos são desenhados para sistemas assíncronos estendidos com esses detectores. [Chandra and Toueg 1996] apresentam o detector $\mathcal{W}$ como a classe mais fraca que possibilita resolver o consenso em sistemas assíncronos, nos quais os processos se comunicam através de troca de mensagens, desde que a maioria dos processos seja correta. Através de uma técnica de redução, eles mostram também que as classes de detectores $\diamondsuit\mathcal{S}$ e Ω são equivalentes a $\mathcal{W}$, representando, portanto, requisitos mínimos para resolução do consenso nesse modelo de sistema.

Vários algoritmos têm sido propostos para o modelo assíncrono estendido com detectores de falhas. A maioria deles considera a comunicação baseada em troca de mensagens. Entretanto, dada a importância desse modelo e analogias entre sistemas de passagens de mensagens e de memória compartilhada, alguns estudos foram dedicados a investigar o problema do consenso em sistemas de memória compartilhada ([Lo and Hadzilacos 1994], [Guerraoui and Raynal 2007], [Delporte-Gallet and Fauconnier 2009]). Informalmente, um sistema com memória compartilhada é um conjunto de processos executando que se comunicam através de um conjunto de células de memória compartilhadas sobre as quais há operações que podem ser executadas por um ou mais processos e que representam a única forma de acessá-las. Para cada célula existe um conjunto de valores possíveis de se armazenar.

Neste trabalho apresentamos um algoritmo genérico para o consenso em sistemas assíncronos de memória compartilhada com processos sujeitos a falhas por colapso (*crashing*). O algoritmo é genérico no sentido de que pode ser instanciado com um detector de falhas que pode ser da classe $\diamondsuit\mathcal{S}$ ou Ω, os quais são os detectores mais fracos que permitem realizar o consenso nesse modelo de sistema. Considerando, portanto, os requisitos mínimos de sincronia necessários para resolver o consenso, nosso algoritmo é ótimo. Ele é ótimo também com relação à resiliência, pois tolera qualquer número de falhas. Ao contrário do que ocorre com o modelo de troca de mensagens, em que uma maioria de corretos é exigida, nosso algoritmo permite que um processo correto termine a execução independente do comportamento dos demais, isto é, ele é *wait-free*.

Recentes avanços na tecnologia de armazenamento apontam para sistemas como SAN–*Storage Area Networks* ou *commodity disks* [Aguilera et al. 2003, Guerraoui and Raynal 2007], nos quais os discos, ao invés de serem controlados por um único processo, são ligados diretamente a uma rede de alta velocidade e acessados dire-

tamente pelos clientes. Em alguns sistemas distribuídos, processos se comunicam através desses discos que implementam assim uma memória compartilhada. Como esses discos são mais baratos do que computadores, são uma opção cada vez mais atrativa para se atingir tolerância a falhas e motivam a proposição de algoritmos como o apresentado aqui, apropriado para tal aplicação. O modelo de memória compartilhada também é comum nas máquinas *multicore* atuais, onde processadores compartilham uma única memória física, ou em sistemas distribuídos onde parte da memória de cada processador (e.g., registradores) é compartilhada por vários processos. Assim, o algoritmo proposto se constitui numa ferramenta fundamental para o desenvolvimento de sistemas confiáveis em tais ambientes.

O resto do artigo está estruturado da seguinte maneira: A Seção 2 traz o modelo do sistema. A Seção 3 apresenta o algoritmo genérico de consenso, cujas provas estão na Seção 4. Na Seção 5 é feita uma discussão sobre o algoritmo proposto e trabalhos relacionados. A Seção 6 conclui o artigo.

2. Preliminares

2.1. Modelo do Sistema

Considera-se um sistema distribuído que consiste de um conjunto finito de $n > 1$ processos $\Pi = \{p_1, ..., p_n\}$, dos quais, f podem falhar por colapso (*crashing*), i.e., parando prematura ou deliberadamente. Após parar, um processo não se recupera. Um processo que não falha em uma execução é dito *correto* e um processo que falha é dito *faltoso*. Os processos trocam informações através da leitura e escrita num arranjo $R[n]$ de registradores *regulares* compartilhados do tipo 1-*escritor/n-leitores* (1W/nR), que se comportam corretamente. 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* de um registrador significa que ele sempre pode executar uma leitura ou escrita e nunca corrompe seu valor.

Um registrador *regular* é mais fraco do que um registrador *atômico* [Lamport 1986]. Num registrador regular, uma leitura não concorrente com uma escrita, retorna o último valor escrito no registrador, e uma leitura que sobrepõe uma ou mais escritas pode obter o último valor escrito antes da leitura iniciar-se ou o valor escrito por qualquer uma das escritas concorrentes. Ou seja, um registrador regular está sujeito a *inversão de valores*. Já um registrador atômico não permite tal inversão. Em um sistema com registradores atômicos, para qualquer execução existe alguma forma de ordenar totalmente leituras e escritas de modo que o valor retornado por uma leitura que sobrepõe uma ou mais escritas seja o mesmo que seria retornado se não houvesse sobreposição.

Não fazemos suposições sobre o tempo que dura cada operação de leitura ou escrita ou mesmo sobre a velocidade dos processos, i.e., o sistema é *assíncrono*. No restante deste artigo, o modelo de sistema definido nesta seção será denominado $\mathcal{AS}[\Pi, f]$.

2.2. Consenso

O problema do consenso é fundamental no projeto de sistemas distribuídos confiáveis. Neste, cada processo p_i propõe um valor v_i e todos os processos têm que decidir por um mesmo valor v. Formalmente, o problema é definido pelas propriedades: (1) Validade – se um processo decide por um valor v, então v é o valor inicial de algum processo;

(2) Acordo uniforme – se um processo decide por um valor v então todos os processos corretos decidem pelo mesmo valor v; (3) Terminação – todos os processos corretos em algum momento acabam por decidir algum valor.

2.3. Detectores de Falhas

Já é bem estabelecido que não existe solução determinística para o consenso em um sistema assíncrono sujeito a falhas [Fischer et al. 1985]. Informalmente, essa impossibilidade é explicada pela dificuldade em distinguir se um processo parou ou está muito lento. Uma solução alternativa é estender o sistema com mecanismos detectores de falhas. Um detector de falhas ($\mathcal{D}$) não confiável é constituído de módulos distribuídos que tem o objetivo de prover o sistema com dicas sobre falhas de processos [Chandra and Toueg 1996]. Cada processo possui um módulo local que funciona como um oráculo fornecendo-lhe, quando requisitado, uma lista de processos considerados suspeitos. Esses módulos podem cometer erros incluindo em suas listas de suspeitos processos corretos ou deixando de incluir processos faltosos. A despeito dessa não-confiabilidade, algoritmos de consenso corretos têm sido propostos para sistemas assíncronos estendidos com esses oráculos [Lo and Hadzilacos 1994, Guerraoui and Raynal 2003].

2.3.1. Detector de Falhas $\diamondsuit\mathcal{S}$

[Chandra and Toueg 1996] classificam os detectores de falhas de acordo com as propriedades completude e acurácia, que os detectores em cada classe exibem. A propriedade completude requer que o $\mathcal{D}$ de fato venha a suspeitar de todo processo faltoso enquanto que a acurácia restringe as suspeições errôneas sobre processos corretos. Os autores então combinam duas propriedades de completude e quatro de acurácia e apresentam oito classes distintas de detectores de falhas. Neste trabalho, o interesse se coloca sobre a classe de detectores de falhas *forte após um tempo* (do inglês, *eventually strong*), ou $\diamondsuit\mathcal{S}$. Um detector $\diamondsuit\mathcal{S}$ satisfaz às seguintes propriedades:

(1) completude forte (*strong completeness*). A partir de algum instante no tempo, todo processo que colapsa será considerado permanentemente suspeito por todo processo correto.
(2) acurácia fraca após um tempo (*eventual weak accuracy*). Existe um instante no tempo a partir do qual algum processo correto jamais será considerado suspeito por qualquer processo correto.

Vale ressaltar que os instantes de estabilidade garantidos pelas propriedades acima não são conhecidos pelos processos. Entretanto, a existência desses instantes permite garantir o término dos algoritmos de consenso baseados em detectores da classe $\diamondsuit\mathcal{S}$.

2.3.2. Detector de Líder Ω

O *detector de líder após um tempo*, conhecido como Ω, também funciona como um oráculo distribuído. O módulo local do detector num processo p_i fornece a identidade de um único processo p_j que p_i considera correto naquele instante. O detector Ω satisfaz à seguinte propriedade:

liderança eventual: existe um instante após o qual o detector fornece a identidade do mesmo processo correto no sistema (i.e., o mesmo líder) para todos os demais processos.

Os detectores $\diamondsuit\mathcal{S}$ e Ω possuem o mesmo poder computacional e são as classes mais fracas de detectores que permitem resolver o consenso em redes assíncronas com Π conhecido, tanto no modelo de passagem de mensagens ([Chandra et al. 1996]), quanto no modelo com memória compartilhada ([Delporte-Gallet et al. 2004]).

3. Algoritmo Genérico para Consenso em Memória Compartilhada

Nesta seção apresentamos um algoritmo genérico para resolver o consenso em um sistema distribuído com memória compartilhada. O algoritmo é genérico porque pode ser instanciado para um sistema estendido com o detector de falhas $\diamondsuit\mathcal{S}$ ou com o detector de líder Ω. Para tanto, o algoritmo define a função $\mathsf{PROPOSITION}_{\mathcal{D}}(e_i, r_i)$ que, a partir de uma consulta ao detector apropriado, tem como objetivo definir a proposta de p_i.

O algoritmo executa em rodadas assíncronas. Cada processo p_i tem acesso às propostas dos demais através da memória compartilhada e tenta tomar uma decisão com base nessas propostas. Entretanto, p_i só decide por um valor v se todas as propostas na memória compartilhada forem iguais a v. Caso contrário, ele parte para uma nova rodada de definição de proposta.

3.1. Memória Compartilhada

A memória compartilhada é constituída de um arranjo $R[n]$ de registradores *regulares* do tipo 1-*escritor-n-leitores* ($1WnR$). O registrador $R[i]$ pertence ao processo p_i, que é o seu único escritor. Entretanto, p_i tem acesso de leitura a qualquer registrador $R[j]$ pertencente a p_j. Cada registrador é composto dos seguintes campos:

(i) $R[i].round$: inteiro que indica a rodada executada por p_i; inicializado com 0.
(ii) $R[i].value$: valor que pode representar uma estimativa, uma proposta ou a decisão do processo; inicializado com $\perp$, denotando um valor padrão que não pode ser proposto por algum processo.
(iii) $R[i].tag$: rótulo que qualifica o valor armazenado em $R[i].value$: est – estimativa; pro – proposta, ou dec – decisão. Inicializado com $\perp$, indica que $R[i].value$ ainda não foi formalizado como estimativa, proposta ou decisão.

As operações que os processos realizam sobre os registradores são as seguintes:

$read(R[i], aux)$: lê o registrador $R[i]$, retornando valor para a variável registro local aux.

$write(R[i], aux)$: escreve o valor na variável registro local aux no registrador $R[i]$.

Embora uma escrita seja executada sobre um registrador (todos os campos), para facilitar a leitura dos algoritmos, algumas vezes expressamos uma escrita de apenas parte dos campos do registrador. Isso é feito sem perda de generalidade, pois como o escritor de cada registrador é único, ele sempre sabe qual foi o último valor escrito. Por exemplo, se a intenção é alterar apenas o campo $value$ de $R[i]$, escrevendo o valor v nele, podemos utilizar a sentença $write(R[i].value, v)$, para denotar a operação:

$$write(R[i].\langle round, value, tag\rangle\ ,\ \langle R[i].round, v, R[i].tag\rangle)$$

3.2. Descrição do Algoritmo de Consenso

O Algoritmo 1 apresenta a função CONSENSUS(v_i) executada por cada processo p_i para decidir por um valor entre os propostos. A entrada v_i é o valor inicial de p_i. O algoritmo funciona em rodadas assíncronas numeradas a partir de 0 (linha 1). O número da rodada é atualizado pela função PROPOSITION$_{\mathcal{D}}(e_i, r_i)$ e depende do oráculo utilizado. Inicialmente, p_i configura sua estimativa para assumir seu valor inicial (linha 1). Então, p_i entra no laço onde permanecerá até que decida um valor (linha 7). Como ao final de alguma iteração do laço a estimativa de p_i pode ser ajustada para $\perp$ (linha 9), ele armazena seu valor corrente de estimativa em v_i para que possa resgatá-lo na próxima iteração(linha 3).

Algorithm 1 CONSENSUS(v_i)

```
(1) r_i := 0;   e_i := v_i;
(2) repeat forever
(3)     if (e_i = ⊥) then e_i := v_i else v_i := e_i;
(4)     PROPOSITION_D(e_i, r_i);
(5)     read (R[1..n] , aux[1..n]);
(6)     proposes := { aux[j].value , ∀ j | aux[j].tag = pro };
(7)     case { (proposes = {v})        then  write (R[i] , ⟨ r_i , v , dec⟩); return (v);
(8)          (proposes = { v , ⊥ })    then  e_i := v; write (R[i] , ⟨ r_i , v , pro⟩);
(9)          (proposes = { ⊥})         then  e_i := ⊥; } endcase
```

A função PROPOSITION$_{\mathcal{D}}(e_i, r_i)$ (linha 4) escreve a proposta de p_i em $R[i]$. p_i então lê o arranjo de registradores (linha 5) e armazena na variável conjunto $proposes$ todas as propostas ali existentes (linha 6). Conforme mostrado adiante, cada $R[i]$ só pode conter, como proposta ($R[i].tag = pro$): um certo valor v, igual para todos os que conseguirem fazer uma proposta válida; ou $\perp$, se o processo não conseguir fazer uma proposta válida. Então, p_i poderá: (i) decidir v, se este for o valor de todas as propostas, e retornar (linha 7); (ii) assumir v como sua estimativa e proposta para a próxima rodada, caso haja propostas iguais a v e a $\perp$, e iniciar a próxima iteração (linha 8); ou (iii) assumir $\perp$ como estimativa para a próxima rodada, caso não haja qualquer proposta válida (linha 9), e assim retomar a estimativa da última rodada (linha 3).

3.3. A Função PROPOSITION$_{\mathcal{D}}(e_i, r_i)$

A função PROPOSITION$_{\mathcal{D}}(e_i, r_i)$ pode ser instanciada com um detector $\diamondsuit\mathcal{S}$ ou Ω. Ela retorna uma proposta v, igual para todos os processos, ou então $\perp$, indicando que p_i não obteve êxito em elaborar uma proposta válida na rodada corrente. Sempre que chamada por CONSENSUS(v_i), PROPOSITION$_{\mathcal{D}}(e_i, r_i)$ executa uma nova rodada cujo número funciona como um relógio lógico, de modo que $r_i > r_j$ indica que um processo p_i que esteja na rodada r_i está mais adiantado, isto é, já executou mais rodadas do que o processo p_j, que está na rodada r_j. Na primeira invocação o valor de e_i é o valor inicial de p_i.

Nas próximas seções são apresentados algoritmos para a função PROPOSITION$_{\mathcal{D}}(e_i, r_i)$ com cada um dos detectores. Além dos registradores compartilhados, os algoritmos usam as seguintes variáveis locais: r_i – número da rodada sendo executada por p_i; e_i – estimativa de p_i; aux – arranjo de registros; a – registro (utilizado como variável auxiliar para a leitura de um registro; possui os campos $\langle r, v, t\rangle$, correspondentes a $\langle round, value, tag\rangle$); c_i – identidade do coordenador da rodada (PROPOSITION$_{\diamondsuit\mathcal{S}}(e_i, r_i)$); l_i – identidade do líder da rodada (PROPOSITION$_{\Omega}(e_i, r_i)$).

3.3.1. Consenso com $\diamondsuit\mathcal{S}$

O Algoritmo 2 (PROPOSITION$_{\diamondsuit\mathcal{S}}(e_i, r_i)$) é uma instância da função de proposição para resolver o consenso com um detector $\diamondsuit\mathcal{S}$. Funciona em rodadas assíncronas sob o paradigma do coordenador rotativo. Cada rodada possui um único coordenador cuja identidade é função do número da rodada que é incrementado de 1 a cada iteração.

Algorithm 2 PROPOSITION$_{\diamondsuit\mathcal{S}}(e_i, r_i)$

```
(1)   r_i := r_i + 1;
(2)   write (R[i], ⟨r_i, e_i, ⊥⟩);
(3)   c_i := (r_i mod n) + 1;        /*calcula a ID do coordenador do round */
(4)   if (c_i = i) then              /*se p_i é coordenador do round */
(5)     read (R[1..n], aux[1..n]); /* verifica estado do registrador
(6)     if (∃ j | aux[j].r > r_i) then   /*se há alguém mais adiantado
(7)       write (R[i], ⟨r_i, ⊥, pro ⟩); /*p_i propõe nada e abandona rodada
(8)     else  if (∃ j | (aux[j].tag = est ∧ aux[j].value ≠ ⊥) then   /*se ∃ estimativa*/
(9)                 e_i := (aux[j].value | ∀j, k: aux[j].tag = aux[k].tag = est,j >= k)
(10)          write (R[i], ⟨r_i, e_i, est⟩);    /* pi divulga sua estimativa (com tag "est") */
(11)          read (R[1..n], aux[1..n]); /* verifica estado do registrador
(12)          if (∃ j | aux[j].r > r_i) then /*se há alguém mais adiantado
(13)            write (R[i], ⟨ r_i, ⊥, pro ⟩); /*p_i propõe nada e abandona rodada
(14)          else  if (∃ j | (aux[j].tag = pro) ∧ (aux[j].value ≠ ⊥)) then  /*se ∃ proposta*/
(15)                    e_i := (aux[j].value | ∀j, k: aux[j].tag = aux[k].tag = pro,j >= k)
(16)                write (R[i] , ⟨ r_i , e_i , pro⟩);    /*p_i divulga sua proposta (tag="pro")*/
(17)  else     /*se pi não é o coordenador da rodada */
(18)    repeat
(19)      read (R[c_i] , a); /*espera proposta do coordenador da rodada ou suspeita */
(20)    until ((a.r > r_i) ∨ (a.t = pro) ∨ (c_i ∈ D_i));
(21)    if ((a.t = pro) ∧ (a.r = r_i)) then /*se proposta válida do coordenador da rodada */
(22)      e_i := a.v; /*adota proposta do coordenador da rodada */
(23)      write (R[i] , ⟨ r_i , e_i , pro ⟩);
(24)  return;
```

Apenas o coordenador propõe um valor na rodada corrente (os demais processos adotam a proposta do coordenador publicando-a em seus registradores). Entretanto, a não confiabilidade inerente aos detectores $\diamondsuit\mathcal{S}$ possibilita a coexistência de coordenadores distintos. Isto porque se qualquer processo p_i, que não é o coordenador corrente, recebe de $\mathcal{D}_i$ a informação de que o coordenador é suspeito (linha 20), embora este permaneça executando (entre as linhas 5 e 16), p_i deixa o *repeat-until* (linhas 18 a 20), retornando para CONSENSUS sem fazer sua proposta. Se nenhuma decisão é tomada, p_i segue para a próxima iteração progredindo para a próxima rodada. Aí, outro coordenador é definido, o qual pode permanecer executando as linhas 5 a 16 junto com ao coordenador anterior.

Dessa forma, se $\mathcal{D}_i$ suspeita erroneamente de coordenadores que não falharam, novos coordenadores podem ser definidos sucessiva e concorrentemente. Nesse sentido, o algoritmo precisa ser indulgente para com o detector, isto é, deve manter a segurança (*safety*) durante períodos de instabilidade e atingir vivacidade quando o sistema se estabiliza [Guerraoui and Raynal 2003].

Descrição do Algoritmo

Parte 1 – Todos os processos

Cada processo p_i começa ajustando o número da rodada (linha 1) e disponibilizando seus valores de rodada e estimativa no registrador compartilhado $R[i]$ (linha 2). Antes de rotular seu registro com "est", esta estimativa de p_i é considerada inválida ($R[i].tag = \perp$). Em seguida, p_i calcula a identidade do coordenador da rodada r_i (linha 3). A partir daí, dois rumos distintos podem ser tomados na Parte 2 (linha 4).

Parte 2a – Processo coordenador da rodada

Se p_i é o coordenador da rodada, vai tentar impor sua proposta. Para isso, p_i lê primeiro os registradores dos demais processos para saber como está o progresso deles (linha 5). Se houver algum processo mais adiantado que ele, p_i desiste de propor um valor e informa o fato escrevendo $\perp$ e "pro", respectivamente, nos campos $value$ e tag de seu registrador (linhas 6-7). Se, no entanto, p_i está na rodada mais adiantada, verifica se já existe alguma estimativa válida ($\neq \perp$) em algum registrador (linha 8). Se houver, p_i assume como sua esta estimativa. Caso contrário, valida sua própria estimativa e a publica (linhas 8-10).

p_i vai então tentar propor sua estimativa (i.e., estabelecê-la como proposta). Para isso ele busca obter as mesmas garantias que obteve para definir sua estimativa: (i) verifica se há algum processo mais adiantado, o que definirá se p_i persistirá no propósito de propor um valor ou não (linhas 11-13); e (ii) verifica se há proposta em algum registrador, o que definirá se sua proposta será igual a alguma pré-existente ou à sua estimativa (linhas 14-15). Se alcançar a linha 16, p_i divulga sua proposta escrevendo em seu registrador um valor diferente de $\perp$ acompanhado do rótulo "pro". Se não tiver alcançado a linha 16, p_i terá escrito o valor $\perp$ acompanhado do rótulo "pro" na linha 7 ou 13, abrindo mão de fazer uma proposta. Em qualquer um dos casos, após escrever seu registrador, p_i retorna para CONSENSUS. A escrita de $\langle -, -, \text{pro} \rangle$ em $R[c_i]$ libera os não-coordenadores da espera nas linhas 18-20, mesmo que o valor escrito em $R[i].value$ seja $\perp$.

Parte 2b – Demais processos: Não coordenadores

Se p_i não é coordenador, tenta obter a proposta do coordenador (linhas 17-20) até que um dos predicados seja verdadeiro: (i) o coordenador publica uma proposta (válida ou não); ou (ii) o coordenador consta da lista de suspeitos de $\mathcal{D}_i$. Se p_i obteve uma proposta do coordenador, p_i assume como sua a proposta do coordenador, divulga-a (linhas 21-23) e retorna para CONSENSUS.

3.3.2. Consenso com Ω

Para esta versão o oráculo utilizado é um detector de falhas da classe Ω. O algoritmo também funciona em rodadas assíncronas, porém aqui apenas o líder progride para uma próxima rodada. Além disso, cada processo, quando líder, executa rodadas com números distintos. Para isso, a primeira rodada executada por um processo p_i tem número igual a i (a identidade de p_i) e as próximas rodadas são calculadas somando-se n ao número da rodada anterior. O resto do algoritmo segue parecido com a versão que usa $\diamondsuit\mathcal{S}$. As diferenças estão, primeiro, no fato de que o líder, diferentemente do coordenador, não é definido em função do número da rodada, e sim da indicação do oráculo Ω (linha 2).

Segundo, o critério de saída da espera nas linhas 18 a 20, é a indicação pelo oráculo Ω de que o líder (corrente até então), deixou de ser líder.

Algorithm 3 PROPOSITION$_\Omega(e_i, r_i)$

```
(1)  if (r_i = 0) then r_i := i;
(2)  l_i := leader();        /*obtém a ID do líder */
(3)  if (l_i = i) then {     /*se p_i é o líder */
(4)     r_i := r_i + n;   write (R[i], ⟨r_i, e_i, ⊥⟩);
(5)     read (R[1..n], aux[1..n]);
(6)     if (∃ j | aux[j].r > r_i) then  /*se há alguém mais adiantado, p_i propõe nada /abandona
(7)        write (R[i], ⟨r_i, ⊥, pro ⟩);
(8)     else  if (∃ j | (aux[j].tag = est ∧ aux[j].value ≠ ⊥) then  /*se ∃ estimativa*/
(9)              e_i := (aux[j].value | ∀j, k: aux[j].tag = aux[k].tag = est, j >= k)
(10)          write (R[i], ⟨r_i, e_i, est⟩);   /* pi divulga sua estimativa (com tag "est") */
(11)          read (R[1..n], aux[1..n]);
(12)          if (∃ j | aux[j].r > r_i) then
(13)             write (R[i], ⟨ r_i, ⊥, pro ⟩);
(14)          else  if (∃ j | (aux[j].tag = pro) ∧ (aux[j].value ≠ ⊥)) then  /*se ∃ proposta*/
(15)                   e_i := (aux[j].value | ∀j, k: aux[j].tag = aux[k].tag = pro, j >= k)
(16)                write (R[i] , ⟨ r_i , e_i , pro⟩);   /*p_i divulga sua proposta (tag="pro")*/
(17) else     /*se pi não é o líder */
(18)    repeat
(19)       read (R[l_i] , l);
(20)    until ((a.r > r_i) ∨ (a.t = pro) ∨ (l_i ≠ leader()));
(21)    if (a.t = pro) then
(22)       e_i := a.v;
(23)       write (R[i] , ⟨ r_i , e_i , pro ⟩);
(24) return;
```

Como no caso dos não coordenadores, os processos não líderes tentam obter a proposta do líder (linhas 17-20), mas aqui a repetição é abandonada se Ω informa uma identidade de processo diferente da informada no início da função (linha 2).

Note que a exemplo do algoritmo anterior, aqui é possível a coexistência de diversos líderes, uma vez que o detector Ω pode retornar identidades distintas para invocações distintas. Da mesma forma, o algoritmo precisa ser indulgente para com o detector, garantindo a segurança (*safety*) durante períodos de instabilidade e atingir vivacidade quando o sistema se estabiliza.

4. Prova de Corretude do Algoritmo de Consenso Genérico

Notação. Considere a seguinte notação para as provas do Consenso: Um processo "p_i propõe um valor v" quando p_i escreve o valor v, bem como o tag = *pro* no seu registrador $R[i]$, ou seja, quando o comando **write** $(R[i], \langle -, v, \text{pro} \rangle)$ é executado por p_i.

Lema 1 *Em um sistema $\mathcal{AS}[\Pi, f]$ estendido com $\mathcal{D}_{\diamond\mathcal{S}}$ ou $\mathcal{D}_\Omega$, se algum processo p_i que invoca PROPOSITION$_\mathcal{D}(-, r_i)$ propõe $v \neq \perp$, então algum processo p_j invocou PROPOSITION$_\mathcal{D}(v, r_j)$, $r_j \leq r_i$.*

Prova. A partir de uma inspeção nos Algoritmos 2 e 3, vemos que um valor $v \neq \perp$ só pode ser proposto por um processo p_i: (A) pelo processo coordenador (linha 16); ou (B)

por um processo não coordenador (linha 23). Neste último caso, o processo não coordenador simplesmente assume a proposta imposta pelo coordenador. Então, é suficiente provar o caso (A). A proposta escrita no registrador (linha 16) é o valor armazenado em e_i. Inicialmente, esse valor é passado como argumento na invocação da função e pode permanecer inalterado até a escrita da proposta. Entretanto, e_i pode ser atualizado antes:

(i) se algum processo p_j tiver escrito uma estimativa em uma rodada anterior $r_j < r_i$ (linhas 8-9). Nesse caso, e_i recebe este valor (ou alguma das estimativas, se houver mais de uma). Note que a primeira estimativa escrita no registrador é, forçosamente, um valor passado como argumento na invocação da função, já que nessa situação, o processo avalia o predicado da linha 8 como $falso$;

(ii) se algum processo p_j tiver escrito uma proposta numa rodada anterior $r_j < r_i$ (linhas 14-15). Nesse caso, e_i recebe este valor (ou alguma das propostas, se houver mais de uma). Note que a primeira proposta escrita no registrador por um processo p_j é, forçosamente, um valor passado como argumento na chamada da função (seja por p_j ou por outro processo), pois se o predicado da linha 14 é avaliado $falso$, a proposta será o próprio e_j recebido na invocação ou uma estimativa obtida conforme (i).

De (A) e (B), o lema segue. □

Lema 2 *Em um sistema $\mathcal{AS}[\Pi, f]$ estendido com $\mathcal{D}_{\Diamond\mathcal{S}}$ ou $\mathcal{D}_{\Omega}$, se algum processo p_i que invoca PROPOSITION$_{\mathcal{D}}(-,-)$ propõe $v \neq \perp$, então $\forall p_j$ que invoca PROPOSITION$_{\mathcal{D}}(-,-)$ e propõe algum valor, propõe $w = v \vee w = \perp$.*

Prova. Vamos considerar especificamente o caso da função PROPOSITION$_{\Diamond\mathcal{S}}$. A prova para o caso da função PROPOSITION$_{\Omega}$ é análoga a esta, bastando fazer a transposição coordenador/líder e das respectivas linhas citadas do algoritmo. Considere dois casos em que um processo p_i faz uma proposta (escreve no registrador $\langle r_i, v, \text{pro}\rangle$):

CASO 1. p_i não é coordenador da rodada. Nesse caso p_i apenas assume a proposta do coordenador (linhas 18 a 23) e portanto o lema se confirma.

CASO 2. p_i é coordenador da rodada. Podemos desmembrar esse caso em dois:

CASO 2A. p_i é o único coordenador da rodada e executa sozinho as linhas 4-16. Trivialmente, se houver proposta válida em algum registrador, é assumida por p_i (linhas 14-15).

CASO 2B. p_i é coordenador da rodada, mas há outros coordenadores: p_i executa o trecho das linhas 4-16 concorrentemente com os demais coordenadores. Isso pode acontecer devido à não confiabilidade de $\mathcal{D}_{\Diamond\mathcal{S}}$ (veja seção 3.3.1). Sem perda de generalidade, suponha que um coordenador p_j executa concorrentemente com p_i e que p_i é o primeiro a propor $v \neq \perp$. Suponha, por contradição, que p_j propõe $w \neq v$. Se ambos propõem valores, ambos executam a linha 16 do algoritmo. Considere os seguintes instantes de tempo:

t_0: instante em que p_i inicia a leitura dos registradores da linha 5;
t_1: instante em que p_j inicia a escrita de $R[i]$ da linha 2;
t_2: instante em que p_j inicia a leitura dos registradores da linha 5; logo $t_1 < t_2$ (A)
t_3: instante em que p_i inicia a escrita de $R[i]$ da linha 10;
t_4: instante em que p_i inicia a leitura dos registradores da linha 11; logo $t_3 < t_4$ (B)

(1) Para p_i propor o valor v na linha 16, pode-se concluir que o predicado da linha 6 foi avaliado *falso*, logo, quando p_i executou a leitura de R[i] da linha 5, p_j ainda não

havia executado a escrita da linha 2 ou, dada a regularidade dos registradores, ambos executaram a leitura e escrita concorrentemente; portanto, temos que $t_0 \leq t_1$.

(2) p_i divulgou sua estimativa executando a escrita de R[i] na linha 10 com $e_i = v$.

(3) Se p_j, pela nossa hipótese, propõe $w \neq v$, na linha 16, então p_j não pode ter atribuído v a e_j na linha 9, isto é, p_j avaliou o predicado da linha 8 como *falso*, donde se conclui que p_j realizou a leitura da linha 5 antes ou concorrentemente à escrita de p_i da linha 10; logo, $t_2 \leq t_3$.

(4) Para que p_i não executasse a escrita da linha 13 (abandonando a chance de fazer sua proposta), deve ter avaliado o predicado da linha 12 *falso*. Logo, nesse instante, quando p_i fez a leitura da linha 11, não havia (ele não pode ter achado) qualquer processo numa rodada mais adiantada que a dele (linha 11). Se esse fosse o caso, a leitura de R_i por p_i teria que ter iniciado antes da escrita de p_j da linha 2; portanto, nesse caso, $t_4 \leq t_1$

De (A), (3), (B) e (4), temos $t_1 < t_2$; $t_2 \leq t_3$; $t_3 < t_4$; $t_4 \leq t_1$ – uma contradição. □

Lema 3 *Em um sistema $\mathcal{AS}[\Pi, f]$ estendido com $\mathcal{D}_{\diamondsuit\mathcal{S}}$ ou $\mathcal{D}_\Omega$, se todo processo p_i invoca PROPOSITION$_\mathcal{D}(v)$, então todo p_i só pode propor v ou $\perp$.*

Prova. A prova é direta de PROPOSITION$_\mathcal{D}$ e do Lema 1. Propostas só podem ser feitas nas linhas 7, 13 ou 16. Nas duas primeiras, o valor proposto é sempre $\perp$. Na última, o valor proposto é $\neq \perp$. Pelo Lema 1, um valor proposto v é tal que algum processo invocou PROPOSITION$_\mathcal{D}(v, -)$. Logo, se todo processo p invoca PROPOSITION$_\mathcal{D}(v, -)$, v é o único valor $\neq \perp$ que pode ser proposto. □

Lema 4 *Em um sistema $\mathcal{AS}[\Pi, f]$ estendido com $\mathcal{D}_{\diamondsuit\mathcal{S}}$ ou $\mathcal{D}_\Omega$, todo processo correto que invoca PROPOSITION$_\mathcal{D}(-,-)$ continuamente, em algum instante propõe um valor $v \neq \perp$.*

Prova. Para este lema, vamos considerar o caso da função PROPOSITION$_{\diamondsuit\mathcal{S}}(e_i, r_i)$. A prova para o caso da função PROPOSITION$_\Omega(e_i, r_i)$ é análoga a esta, bastando fazer a transposição coordenador/líder, considerando as respectivas linhas do algoritmo. Além disso, deve-se considerar a propriedade de liderança eventual de Ω.

A cada rodada é definido um coordenador em função do número da rodada (linha 3). Enquanto o coordenador tenta impor uma proposta (linhas 5 a 16), os não coordenadores aguardam para obter a proposta do coordenador (linhas 18 a 20). Uma vez que o coordenador escreva sua proposta no registrador (linha 16), os demais processos obtém e assumem essa proposta (linhas 19 a 23). Entretanto, se algum processo suspeita que o coordenador falhou (linha 20), retorna da função sem fazer uma proposta (linhas 21-24).

Da propriedade acurácia fraca após um tempo do detector de falhas $\diamondsuit\mathcal{S}$, existe um instante a partir do qual algum processo correto jamais será considerado suspeito por qualquer processo correto. Então, existe um instante a partir do qual um processo correto que é coordenador fará uma proposta v (linha 16) e os demais processos corretos obterão e assumirão essa proposta (linhas 19 a 23). □

Lema 5 *Em um sistema $\mathcal{AS}[\Pi, f]$ estendido com $\mathcal{D}_{\diamondsuit\mathcal{S}}$ ou $\mathcal{D}_\Omega$, ao invocar CONSENSO$(-)$, se processo p_i decide por um valor v, então qualquer processo p_j que decide, decide v.*

Prova Acordo uniforme: Para que um processo p_i decida na linha 7, ele tem que ter encontrado todas as propostas no arranjo de registradores iguais a v (linhas 5-7). Pelo Lema 2, se algum processo que invoca PROPOSITION$_{\mathcal{D}}(-,-)$ propõe um valor $v \neq \perp$, então todos os demais processos que invocam PROPOSITION$_{\mathcal{D}}(-,-)$ e propõem um valor, propõem v ou $\perp$. Logo, nenhum processo p_j irá propor e escrever no seu registrador um valor w, $v \neq w \neq \perp$. Portanto, se p_j decide na linha 7, ele decide v. □

Lema 6 *Em um sistema $\mathcal{AS}[\Pi, f]$ estendido com $\mathcal{D}_{\diamondsuit\mathcal{S}}$ ou $\mathcal{D}_{\Omega}$, ao invocar CONSENSO$(-)$, um processo correto p_i termina por decidir um valor v.*

Prova Terminação: O processo p_i decide ao executar a linha 7. Pelo Lema 2, se um processo propõe v, todas as propostas diferentes de $\perp$ são iguais a v. Pelo Lema 4, todo processo correto que invoca PROPOSITION$_{\mathcal{D}}(-,-)$ continuamente, em algum instante propõe um valor $v \neq \perp$. Então, todo processo correto p_i acaba por satisfazer o predicado da linha 7 e decide v. □

Lema 7 *Em um sistema $\mathcal{AS}[\Pi, f]$ estendido com $\mathcal{D}_{\diamondsuit\mathcal{S}}$ ou $\mathcal{D}_{\Omega}$, ao invocar CONSENSO$(-)$, se um processo p_i decide v, então v é o valor inicial de algum processo.*

Prova Validade: Os únicos valores de entrada no algoritmo são os valores iniciais dos processos passados na invocação de CONSENSUS(v_i) por p_i. Esses mesmos valores são passados para PROPOSITION$_{\mathcal{D}}(e_i, r_i)$ em sua primeira invocação (linha 4). Então, qualquer estimativa e proposta escrita nos registradores na chamda de PROPOSITION$_{\mathcal{D}}(e_i, r_i)$ é uma cópia de um desses valores iniciais (linhas 8-10, 14-16). Assim, um valor v decidido é um dos valores propostos inicialmente por algum processo (linhas 5-7).

Teorema 1 *Em um sistema $\mathcal{AS}[\Pi, f]$ estendido com $\mathcal{D}_{\diamondsuit\mathcal{S}}$ ou $\mathcal{D}_{\Omega}$, o Algoritmo 1 satisfaz às propriedades do Consenso (definidas em 2.2).*

Prova. Segue diretamente dos lemas 5, 6 e 7. □

Teorema 2 *O Algoritmo 1 é wait-free.*

Prova. Mostramos que o algoritmo é correto a despeito de $(n-1)$ falhas. Suponha um instante t em uma execução em que $(n-1)$ processos falham. Seja p_i o único processo que não falha nesta execução. Ao executar PROPOSITION$_{\mathcal{D}}(-,-)$, em algum instante p_i virá a ser coordenador (ou líder) e não abandona precipitadamente sua execução sem fazer proposta, já que não haverá qualquer processo em uma rodada maior (linhas 6-7 e 12-13). Pelo contrário, virá a propor um valor v e ao retornar, decidirá por este valor. □

5. Discussão e Trabalhos Relacionados

Considerando que os detectores $\diamondsuit\mathcal{S}$ ou Ω são os mais fracos que permitem o consenso, o algoritmo proposto é ótimo com relação aos requisitos de sincronia. Ele também é ótimo com relação à resiliência, uma vez que ele é *wait-free* – admite (n-1) faltas, qualquer processo termina em um número limitado de passos, independente do comportamento dos demais. Quanto ao custo, o algoritmo também é ótimo [Lo and Hadzilacos 1994]. São necessários apenas n registradores regulares, do tipo 1-*escritor-n-leitores*. Esses registradores são mais fracos, por um lado, do que registradores atômicos, e por outro, do que registradores *n-escritores-n-leitores*. Além disso, se o detector ($\diamondsuit\mathcal{S}$ ou Ω) se comporta perfeitamente, na primeira rodada o algoritmo converge.

O problema do consenso é amplamente estudado e vários algoritmos têm sido propostos, a maioria deles para sistemas em que os processos se comunicam através de passagem de mensagens [Chandra and Toueg 1996, Guerraoui and Raynal 2003]. Para o modelo de memória compartilhada, poucos são os trabalhos identificados, além do nosso [Lo and Hadzilacos 1994, Delporte-Gallet et al. 2004, Guerraoui and Raynal 2007, Khouri et al. 2012].

[Lo and Hadzilacos 1994] propõem um algoritmo de consenso para um sistema assíncrono. Entretanto, eles utilizam registradores *atômicos*, enquanto que nós utilizamos registradores regulares, que são mais fracos. Eles provam que são necessários pelo menos n registradores (atômicos) 1-*escritor/n-leitores* ($1WnR$) para construir algoritmos de consenso *wait-free*, usando um detector de falha da classe *Strong*. O nosso algoritmo funciona corretamente com n registradores $1WnR$ mais fracos – regulares, usando um detector mais fraco ($\diamondsuit\mathcal{S}$ ou Ω).

[Guerraoui and Raynal 2003] identificam a estrutura da informação dos algoritmos de consenso indulgentes para com seus oráculos e estudam a complexidade inerente a estes. Eles apresentam um algoritmo de consenso genérico que pode ser instanciado com um oráculo específico e mantém a mesma complexidade segundo alguns critérios. Eles consideram um modelo assíncrono sujeito a falhas, onde os processos se comunicam através de passagem de mensagens e o número máximo de falhas deve ser $f < n/2$.

[Guerraoui and Raynal 2007] apresentam um arcabouço que unifica uma família de algoritmos de consenso baseado no detector de falhas Ω. O algoritmo pode ser configurado para modelos de comunicação diferentes: memória compartilhada, rede de área compartilhada, passagem de mensagens e sistemas de discos ativos, mas em qualquer desses modelos, o oráculo considerado é o Ω.

Trilhando um caminho semelhante ao de [Guerraoui and Raynal 2003, Guerraoui and Raynal 2007], nós propomos um algoritmo genérico que permite configurar o detector de falhas a ser utilizado–$\diamondsuit\mathcal{S}$ ou Ω, em um modelo de memória compartilhada. Enquanto o algoritmo proposto em [Guerraoui and Raynal 2003] é restrito a sistemas com $f < n/2$ (número de processos faltosos inferior à metade do número de participantes), nós mostramos que no modelo de memória compartilhada, é possível obter algoritmos indulgentes com uma resiliência $n - 1$.

[Khouri et al. 2012] propõem um protocolo para consenso num sistema dinâmico (o conjunto de processos é desconhecido). Sua abordagem consiste em usar a abstração *detector de participantes* para construir o *membership* do sistema e, conforme o grafo da conectividade do conhecimento, aplicar um algoritmo clássico para a realização do consenso. Com uma pequena adaptação para contar o número de processos participantes durante o conhecimento do sistema, o algoritmo aqui apresentado pode ser utilizado neste protocolo e resolver o consenso num sistema dinâmico de memória compartilhada.

6. Conclusão

Neste artigo apresentamos um algoritmo genérico para a resolução do consenso num sistema assíncrono sujeito a falhas que pode ser instanciado com um detector $\diamondsuit\mathcal{S}$ ou Ω. O algoritmo proposto pode ser aplicado a sistemas distribuídos em que processadores compartilham parte de sua memória entre vários processos, máquinas *multicore* atuais, ou

sistemas como *Storage Area Networks*, utilizados para o compartilhamento de armazenamento e que implementam uma memória compartilhada.

Mostramos que é possível construir tal algoritmo, para o modelo proposto, usando n registradores regulares. Este é o número mínimo de registradores atômicos necessários definido na literatura [Lo and Hadzilacos 1994] para qualquer algoritmo de consenso *wait-free* em um modelo estendido com um detector mais forte que o adotado por nós. Consideramos para um próximo trabalho investigar a possibilidade de construir algoritmos de consenso *wait-free* utilizando apenas registradores *safe*, os quais retornam o valor lido, se a leitura não sobrepõe alguma escrita; porém podem retornar qualquer valor do domínio se a leitura for concorrente a uma ou mais escritas [Lamport 1986].

References

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.

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.

Delporte-Gallet, C. and Fauconnier, H. (2009). Two consensus algorithms with atomic registers and failure detector omega. In *Proceedings of the 10th International Conference on Distributed Computing and Networking*, ICDCN '09, pages 251–262, Berlin, Heidelberg. Springer-Verlag.

Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., Hadzilacos, V., Kouznetsov, P., and Toueg, S. (2004). The weakest failure detectors to solve certain fundamental problems in distributed computing. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, PODC '04, pages 338–346, New York, NY, USA. ACM.

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.

Guerraoui, R. and Raynal, M. (2003). The information structure of indulgent consensus. *IEEE Transactions on Computers*, 53:2004.

Guerraoui, R. and Raynal, M. (2007). The alpha of indulgent consensus. *Comput. J.*, 50(1):53–67.

Khouri, C., Greve, F., and Tixeuil, S. (2012). Consenso com participantes desconhecidos em memoria compartilhada. 30o SBRC, Porto Alegre, RS, Brasil. SBC.

Lamport, L. (1986). On interprocess communication. *Distributed Computing*, 1(2):77–101.

Lo, W.-K. and Hadzilacos, V. (1994). Using failure detectors to solve consensus in asynchronous shared-memory systems (extended abstract). In *Proceedings of the 8th International Workshop on Distributed Algorithms*, WDAG '94, pages 280–295, London, UK. Springer-Verlag.

Loui, M. C. and Abu-Amara, H. H. (1987). Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183.

Um Espaço de Tuplas Tolerante a Intrusões sobre P2P

Davi da Silva Böger[1], **Eduardo Alchieri**[2], **Joni da Silva Fraga**[1]

[1]DAS - Universidade Federal de Santa Catarina (UFSC)
[2]CIC, Universidade de Brasília (UnB)

Resumo. *O uso de espaços de tuplas tem se mostrado uma solução atrativa para coordenação entre processos em sistemas distribuídos abertos e dinâmicos, como redes P2P, principalmente pelas suas características de desacoplamento espacial e temporal. Nestes ambientes, caracterizados como sistemas heterogêneos e abertos, aumenta significativamente a possibilidade de processos maliciosos estarem presentes em determinada computação. Neste sentido, este trabalho apresenta nossos esforços na concretização de um Espaço de Tuplas (ET) sobre um overlay P2P, que tolera a presença de processos maliciosos. O ET é construído sobre uma infraestrutura para construção de memórias compartilhadas dinâmicas e tolerantes a intrusões, descrita em outro trabalho.*

1. Introdução

As redes par a par (**peer-to-peer**, P2P) são uma arquitetura interessante para sistemas distribuídos, pois permitem o uso eficiente dos recursos ociosos disponíveis na Internet e têm a capacidade de aumentar o número de nós sem detrimento do desempenho. Algumas redes P2P oferecem primitivas de comunicação com latência e número de mensagens de ordem logarítmica em relação ao número de nós [Rowstron and Druschel 2001]. Essa escalabilidade permite que recursos disponíveis em uma grande quantidade de nós possa ser utilizado de maneira vantajosa.

Apesar das vantagens, as redes P2P apresentam desafios para o provimento de confiabilidade. Essas redes normalmente são formadas dinamicamente por nós totalmente autônomos que podem entrar e sair do sistema a qualquer momento. Tal dinamismo dificulta a manutenção da consistência das informações distribuídas no sistema, bem como a construção de aplicações mais complexas que poderiam se beneficiar da escalabilidade [Baldoni et al. 2005]. A grande maioria dos sistemas P2P são aplicações de disseminação de informações pouco mutáveis ou autoverificáveis. Além disso, essas redes não possuem gerência global, são redes de pares com grande abertura. Logo, as redes P2P podem conter nós maliciosos que colocam em risco o funcionamento das aplicações [Wallach 2003].

Um Espaço de Tuplas [Gelernter 1985] (ET) é um mecanismo de coordenação de processos que apresenta desacoplamento espacial, isto é, processos não precisam conhecer os endereços um do outro para se comunicar; e desacoplamento temporal, isto é, processos não precisam estar ativos simultaneamente para efetivar a comunicação. Essas propriedades são garantidas pelo uso de uma memória associativa na qual as mensagens persistem e podem ser acessadas por buscas baseadas no conteúdo. Essas mensagens persistentes e endereçáveis por conteúdo são denominadas **tuplas** e a memória que as armazena é denominada de **espaço de tuplas**. Os espaços de tuplas são especialmente interessantes em sistemas abertos e dinâmicos, como as redes P2P, nos quais processos não possuem conhecimento completo sobre os demais participantes do sistema. Nós podem se comunicar armazenando tuplas em um ET global que garante a persistência dessas informações, mesmo após a saída do nó que produziu a tupla.

Neste sentido, este trabalho descreve a construção de um espaço de tuplas tolerante a intrusões, que executa sobre uma rede P2P. O espaço de tuplas funciona sobre uma infraestrutura para suporte a aplicações de memória compartilhada tolerantes a intrusões definida em [Böger et al. 2012]. Essa infraestrutura divide um **overlay** P2P estruturado em um conjunto de segmentos e fornece operações para encontrar e acessar segmentos, que são dinâmicos e suportam a entrada e saída de nós, executando um protocolo de Replicação Máquina de Estados (RME) reconfigurável [Lamport et al. 2010].

Para que o ET proposto possa fazer uso dessa segmentação, é necessário cumprir com a exigência relacionada à indexação do estado da aplicação, que deve ser feita de forma a permitir a divisão e união de partes do estado de acordo com as divisões e uniões dos segmentos do sistema. De maneira geral, um espaço de tuplas é um multiconjunto de tuplas, isto é, uma coleção de tuplas que permite repetição de elementos idênticos. Indexamos o ET atribuindo uma chave a cada tupla e dividimos o mesmo em espaços menores contendo parte das tuplas com chaves numericamente próximas. Cada segmento S, conforme definido na camada de segmentação, armazena as tuplas com chaves dentro do intervalo $K(S)$. Para atribuição das chaves, utilizamos um esquema de indexação multidimensional baseado em **Curvas de Preenchimento de Espaço** (**Space Filling Curves** ou SFCs em inglês), em especial a **Curva de Hilbert** [Lawder and King 2000]. A Curva de Hilbert apresenta a propriedade de ter boa localidade, isto é, tuplas com conteúdo próximo tendem a possuir índices próximos, o que facilita a realização de buscas por conteúdo.

O ET proposto neste artigo, por ser baseado na infraestrutura descrita em [Böger et al. 2012], apresenta a propriedade de garantir a consistência das informações mesmo na presença de nós maliciosos. Isso assegura o funcionamento correto do ET, porém não provê características de segurança como privacidade de tuplas e controle de acesso. Existem maneiras de garantir essas propriedades de segurança adicionais [Bessani et al. 2008], porém isso está fora do escopo deste trabalho.

O restante do artigo está dividido da seguinte forma: a Seção 2 apresenta o conceito de espaços de tuplas; a Seção 3 apresenta o modelo de sistema; a Seção 4 descreve as curvas de preenchimento de espaço e a curva de Hilbert; a Seção 5 apresenta a construção do espaço de tuplas propriamente dito e a Seção 6 discute as soluções propostas; finalmente, a Seção 7 aborda alguns trabalhos relacionados e a Seção 8 conclui o artigo.

2. Espaços de Tuplas

Um espaço de tuplas (ET) pode ser visto como um objeto de memória compartilhada que permite a interação entre processos distribuídos [Gelernter 1985]. Neste espaço, estruturas de dados genéricas chamadas de **tuplas**, podem ser inseridas, lidas e removidas. Uma tupla t é uma sequência ordenada de campos $\langle f_1 ... f_n \rangle$, onde cada campo f_i que contém um valor é dito *definido*. Uma tupla onde todos os campos são definidos é chamada de **entrada**. Uma tupla $\bar{t}$ é chamada de **molde** se alguns de seus campos não possuem valores definidos. Um espaço de tuplas somente pode armazenar entradas, nunca moldes. Os moldes são usados para acessar as tuplas do espaço. Diz-se que uma entrada t e um molde $\bar{t}$ **combinam** ($t \leq_m \bar{t}$) se e somente se: (i) ambos têm o mesmo número de campos, e (ii) todos os valores dos campos definidos em t possuem o mesmo valor dos campos correspondentes em $\bar{t}$. Por exemplo, uma tupla $\langle \text{WTF}, \text{BSB}, 2013 \rangle$ combina com o molde $\langle \text{WTF}, *, 2013 \rangle$ (onde '$*$' denota um campo não definido do molde).

Um espaço de tuplas provê três operações básicas [Gelernter 1985]: $out(t)$ que

adiciona uma tupla t no espaço de tuplas; $in(\bar{t})$ que lê e remove, do espaço de tuplas, uma tupla t que combine com o molde $\bar{t}$; e $rd(\bar{t})$ que tem um comportamento similar ao da operação in, mas que somente faz a leitura da tupla combinando com $\bar{t}$, sem removê-la do espaço. As operações in e rd são bloqueantes, i.e., se nenhuma tupla que combine com o molde $\bar{t}$ está disponível no espaço de tuplas, o processo fica bloqueado até que uma se faça disponível. Uma extensão típica deste modelo é a provisão de variantes não bloqueantes das operações de leitura: inp e rdp [Gelernter 1985]. Estas operações funcionam exatamente como suas versões bloqueantes, a não ser pelo fato que retornam um valor de erro quando uma tupla que combine com o molde não esteja disponível no espaço de tuplas. Uma característica importante do espaço de tuplas é a natureza associativa do acesso: as tuplas não são acessadas por um endereço ou identificador, mas sim pelo seu conteúdo.

3. Modelo de Sistema

Consideramos um sistema distribuído formado por um conjunto infinito Π de processos (ou nós), interconectados por enlaces de comunicação (**links**) formando desta maneira uma rede. Cada nó possui um endereço único de rede e pode enviar mensagens para qualquer outro nó, desde que conheça seu endereço. Um nó correto sempre age de acordo com a especificação dos seus protocolos. Um nó malicioso (ou bizantino [Lamport et al. 1982]) não se comporta segundo as especificações, agindo de maneira arbitrária. Assume-se que em qualquer momento da execução, no máximo f nós faltosos estão presentes no sistema. Este parâmetro é global e conhecido por todos os nós.

A infraestrutura subjacente ao espaço de tuplas, definida em [Böger et al. 2012], está dividida em quatro camadas, conforme ilustrado na Figura 1. A camada inferior é a rede, que oferece canais ponto a ponto confiáveis entre quaisquer pares de nós. O atraso na entrega das mensagens e as diferenças de velocidades entre os nós do sistema respeitam um modelo de sincronia parcial [Dwork et al. 1988], no qual é garantido a terminação de protocolos de Replicação Máquina de Estados que são usados nas camadas superiores. No entanto, não há garantia de sincronismo por toda a execução. Imediatamente acima da rede, encontram-se duas camadas independentes que são usadas para construir a camada de segmentação: a camada de **overlay** implementa uma rede P2P estruturada, no estilo anel (p. ex. [Rowstron and Druschel 2001]), sobre a camada de rede, com busca de nós distribuídos eficiente e tolerante a intrusões (em [Böger et al. 2012], o **overlay** definido em [Castro et al. 2002] é utilizado); e a camada de suporte à replicação fornece uma abstração de Replicação Máquina de Estados (RME) reconfigurável (em [Böger et al. 2012], a estratégia definida em [Lamport et al. 2010] é assumida) usada para garantir a disponibilidade e consistência das informações contidas no sistema. Em [Böger et al. 2012], tanto as premissas de sistema quanto as funcionalidades das camadas inferiores são apresentados em maior detalhe.

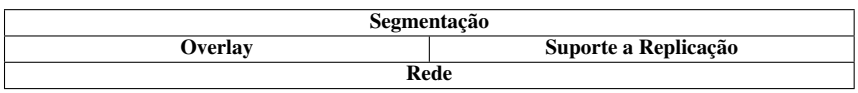


Figura 1. Camadas do sistema.

A camada de segmentação divide o anel lógico do **overlay** em segmentos compostos de nós contíguos, sendo cada segmento responsável por um intervalo de chaves do **overlay**. Para fins de disponibilidade, todos os nós do mesmo segmento armazenam o

mesmo conjunto de dados replicados. A consistência desses dados é mantida usando Replicação Máquina de Estados reconfigurável, provido pela camada de suporte à replicação.

Os segmentos são dinâmicos, ou seja, suas composições podem mudar com o tempo a partir da entrada e saída de nós. Cada segmento é descrito por um certificado de segmento (S), que contém a lista dos nós membros ($S.members$), o intervalo de chaves do **overlay** que cabe ao segmento ($K(S) = [S.start, S.end)$) e um contador de reconfigurações ($S.confId$). A cada reconfiguração, um novo conjunto de nós (denominado composição ou visão) passa a participar no segmento e a executar os algoritmos associados de suporte à RME, bem como um novo certificado de segmento é gerado e assinado pelos membros do segmento antigo, garantindo a autenticidade do mesmo. O número de nós de um segmento pode aumentar ou diminuir, porém, para manter o desempenho da RME, esse número é mantido dentro de um certo intervalo $[n_{MIN}, n_{MAX}]$. Isso é garantido pela divisão de segmentos grandes e pela união de segmentos pequenos adjacentes.

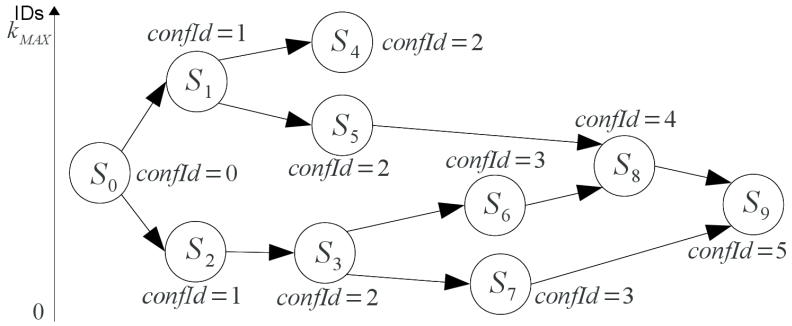


Figura 2. Dinâmica da segmentação em uma possível execução

A Figura 2 ilustra as reconfigurações ocorridas em uma possível execução. Com base nessa execução, três situações de reconfiguração distintas são descritas a seguir:

(1) O segmento inicial S_0 inicia a execução sendo responsável por todas as chaves do **overlay**, ou seja, $K(S_0) = [0, k_{MAX}]$, onde k_{MAX} é a maior chave possível. S_0 possui $confId = 0$, indicando que é o segmento inicial. Quando o número de nós de S_0 aumenta para além de n_{MAX}, ocorre a divisão, na qual são gerados os segmentos S_1 e S_2, ambos com $confId = 1$. Os membros de S_0, juntamente com os membros que recém entraram no sistema, são divididos entre os novos segmentos, de forma que S_1 e S_2 possuam ambos um número de nós maior ou igual a n_{MIN};

(2) A partir do segmento S_2, se um certo número de nós entrar e sair do segmento de forma que o número de nós se mantém estável, a reconfiguração não precisa realizar união ou divisão. Dessa forma, o conjunto de chaves cobertas pelo segmento não se altera pela reconfiguração, ou seja, $K(S_3) = K(S_2)$. O parâmetro $confId$ é incrementado, de forma que $S_3.confId = S_2.confId + 1 = 2$;

(3) Partindo do segmento S_6, pode ocorrer de uma parcela grande dos nós do segmento sair do sistema. Isso leva S_6 a iniciar uma união a fim de evitar terminar a reconfiguração com menos de n_{MIN} nós. Para tal é utilizado o segmento sucessor de S_6, isto é, o segmento S_5. O conjunto de membros resultantes da união é formado pelos membros dos dois segmentos, levando em conta pedidos de entrada e saída de ambos. O conjunto de chaves do segmento resultante S_8 é dado pela união dos conjuntos de chaves de S_5 e S_6, ou seja, $K(S_8) = K(S_5) \cup K(S_6)$. O valor de $confId$ é o incremento do maior valor nos segmentos S_5 e S_6, de forma que $S_8.confId = \max\{S_5.confId, S_6.confId\} + 1 = 4$.

A camada de segmentação apresenta as seguintes operações:

• $SegJoin(C_p)$ realiza a entrada do nó p na camada de segmentação por meio da apresentação do certificado de nó C_p. Esse certificado é o mesmo usado na camada de **overlay**;

• $SegLeave(C_p)$ trata da saída do nó p. C_p é o certificado apresentado na entrada;

• $SegFind(k, k')$ busca todos os certificados de segmento responsáveis por chaves no intervalo $[k, k']$. Os certificados de segmentos encontrados são repassados à camada superior por meio da chamada $SegFindOk$. A operação garante que todo o intervalo de busca é coberto pelos segmentos encontrados;

• $SegRequest(S_q, req)$ envia a requisição req ao segmento S_q usando a camada de suporte à replicação e retorna a resposta recebida da ME. Um temporizador é usado para interromper a requisição caso S_q seja obsoleto. Nesse caso, a operação retorna um valor distinto $\perp$. Caso contrário, a operação $SegDeliver(C_p, req)$ entrega req conforme a ME do segmento de destino. Atrasos na rede podem provocar o estouro do temporizador, mesmo se S_q for atual. Isso deve ser tratado na camada superior;

• $SegResponse(C_p, resp)$ envia a resposta ao cliente através da camada de rede;

• $SegNotify$ é bastante simples, consistindo no envio direto de uma mensagem a um nó, utilizando para isso a camada de rede. No nó cliente, qualquer mensagem recebida de pelo menos $f + 1$ nós de um mesmo segmento é repassada com $SegNotifyDeliver$;

• $SegReconfigure()$ operação interna da camada de segmentação, executada em intervalos de tempo determinados, que efetiva as entradas/saídas de nós conforme as chamadas de $SegJoin$/$SegLeave$. Nessa operação uniões e divisões de segmentos são efetivadas;

• $SegGetAppState$ e $SegSetAppState$ servem, respectivamente, para obter e alterar o estado da aplicação que executa acima da camada de segmentação.

4. Curvas de Preenchimento de Espaço

Uma curva de preenchimento de espaço (**Space Filling Curve**, SFC) é um mapeamento bidirecional entre pontos de um hipercubo de D dimensões e pontos de uma linha, ou seja, é uma curva que passa por todos os pontos de um hipercubo D-dimensional exatamente uma vez [Lawder and King 2000]. A Figura 3 (b) mostra um exemplo simples de uma SFC sobre um quadrado duas dimensões e coordenadas entre $(0, 0)$ e $(3, 3)$. Cada ponto do espaço é representado por um quadrado e a curva passa exatamente uma vez no centro de cada um desses quadrados.

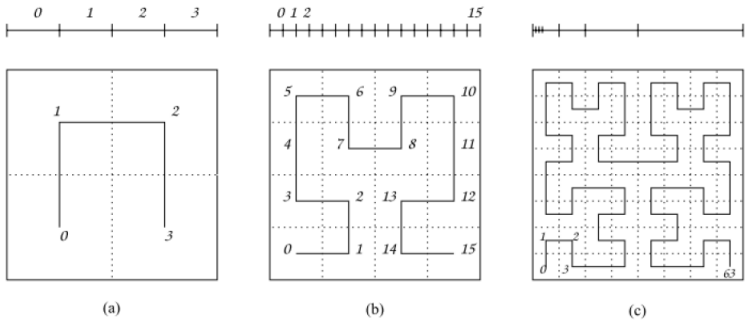


Figura 3. Desenho da curva de Hilbert bidimensional nas três primeiras ordens

Uma SFC pode ser usada como mecanismo de indexação de tuplas em um ET. Podemos tratar as tuplas como elementos de um espaço multidimensional, cada campo da tupla sendo uma coordenada, e podemos usar uma SFC para calcular um índice (ou

chave) associado à tupla. Usando o exemplo da Figura 3 (b), podemos indexar tuplas de dois campos com valores numéricos entre 0 e 3: a tupla $\langle 0, 0\rangle$ tem índice 0, a tupla $\langle 1, 0\rangle$ tem índice 1, a tupla $\langle 1, 1\rangle$ tem índice 2, e assim sucessivamente. Esse exemplo é simplificado e não permite a indexação de tuplas em situações mais complexas. A Seção 5 mostra uma generalização desse princípio que permite indexação de qualquer tupla.

Dado um mesmo espaço multidimensional, existem diversas formas de se indexar os pontos desse espaço, ou seja, existe mais de uma SFC que pode ser aplicada, e cada indexação apresenta propriedades distintas. A curva de Hilbert é um tipo de SFC que apresenta propriedades de localidade, no sentido de que pontos adjacentes na curva unidimensional são também adjacentes no espaço multidimensional [Lawder and King 2000]. Essa localidade facilita as buscas por conteúdo necessárias em um ET, uma vez que tuplas que casem com o mesmo molde tem chance maior de terem índices próximos. Usando ainda o exemplo da Figura 3 (b), podemos ver que uma busca usando o molde $\langle 1, *\rangle$ precisa localizar tuplas com índices nos intervalos $[1, 2]$ e $[6, 7]$.

De maneira geral, dado um espaço multidimensional de D dimensões, uma curva de Hilbert de ordem k divide o espaço em 2^{Dk} subespaços iguais e trata o centro desses subespaços como os pontos pelos quais a curva deve passar. Dessa forma, uma curva de ordem k divide cada eixo do espaço em k níveis e possui 2^{Dk} pontos. A Figura 3 ilustra as três primeiras ordens de uma curva de Hilbert em duas dimensões. A curva de primeira ordem (a) divide o quadrado em quatro partes e percorre os quadrantes em sentido horário. A curva de segunda ordem (b) é construída a partir da curva de primeira ordem pela substituição de cada quadrante por uma curva de primeira ordem rotacionada. A curva de terceira ordem (c) é obtida de maneira similar. Em geral, a curva de ordem $k + 1$ é construída a partir da curva de ordem k dividindo cada ponto desta curva em um subespaço de 2^D pontos e traçando uma curva de ordem 1 rotacionada de forma a se conectar com os subespaços vizinhos. Mais detalhes sobre o procedimento de cálculo podem ser encontrados em [Lawder and King 2000].

A curva completa é obtida aplicando a definição recursiva infinitamente, porém para a construção de índices são usadas as curvas de ordem finita. A ordem da curva está relacionada à quantidade de informação (número de bits) necessária para representar as coordenadas dos pontos do espaço e dos pontos da curva unidimensional. Em geral, dado um espaço D-dimensional e uma curva de k-ésima ordem, cada coordenada dos pontos do espaço precisam de k bits e um ponto qualquer (tanto no espaço multidimensional quanto na curva unidimensional) precisa de $D \times k$ bits. Assumimos que, dado um ponto x no espaço multidimensional, $Hilbert(x)$ representa o ponto na curva de Hilbert correspondente a x.

5. Espaço de Tuplas Tolerante a Intrusões

A camada de espaço de tuplas, construída sobre a camada de segmentação, implementa as operações out, rd, rdp, in e inp. A indexação do espaço de tuplas, necessária para a camada de segmentação, é realizada pela atribuição de chaves a tuplas individuais e de conjuntos de chaves a moldes. De maneira geral, se uma tupla t possui chave k, então qualquer molde $\bar{t}$, tal que $t \leq_m \bar{t}$, deve possuir um conjunto de chaves $\bar{K}$ tal que $k \in \bar{K}$. Dessa maneira, se a tupla t é armazenada no segmento responsável por k, então uma busca cobrindo $\bar{K}$ termina por encontrar t. Esta estratégia não necessita que chaves associadas às tuplas sejam únicas, i.e., tuplas diferentes podem possuir a mesma chave. Basta que a chave associada à tupla aponte sempre para o segmento que a armazena a tupla.

A Curva de Hilbert é utilizada para atribuir chaves a tuplas e conjuntos de chaves a moldes, de forma a possibilitar que as tuplas inseridas no ET sejam posteriormente encontradas. Seja D um parâmetro global do sistema que indica o número máximo de dimensões de uma tupla. A **chave** relacionada a uma tupla $t = \langle t_1, \ldots, t_l \rangle$, onde $l \leq D$, é calculada da seguinte forma: a tupla é normalizada para $D + 1$ dimensões na forma $t' = \langle l, t_1, \ldots, t_l, \perp_{l+1}, \ldots, \perp_D \rangle$. Na sequência, é calculado o **hash** (ex. SHA-1) de cada campo $t''_i = hash(t'_i)$, $\forall i \in [1, D + 1]$; a tupla $t'' = \langle t''_1, \ldots, t''_{D+1} \rangle$ é tratada como um ponto no espaço $D + 1$-dimensional e é mapeada em um índice pela função $k = Hilbert(t'')$. Esse procedimento é representado pela função $TupleKey(t)$. Para inserir a tupla t, calcula-se a chave $k = TupleKey(t)$ e insere-se a tupla no segmento responsável por k usando a camada de segmentação.

Note que é possível que tuplas distintas possuam a mesma chave, portanto $k \in \bar{K}$ não necessariamente implica que $t \leq_m \bar{t}$. Colisões de chaves de tuplas podem resultar de colisões dos **hashes** das coordenadas ou de truncamentos realizados pela função $Hilbert$, uma vez que esta recebe D parâmetros com 128 bits cada (se for usado SHA-1) e retorna um número do mesmo tamanho dos identificadores do **overlay**. Se o número de bits do identificador do **overlay** é menor que D multiplicado pelo número de bits da função **hash** utilizada, então alguma informação é perdida pela função $Hilbert$ de maneira que colisões podem ocorrer. No entanto, como a camada de segmentação permite a execução de requisições arbitrárias, podemos lidar com colisões enviando o próprio molde ao segmento responsável pelas chaves cobertas na busca. Dessa forma, as tuplas são encontradas localmente usando o molde e não somente a chave, logo apenas tuplas que efetivamente casam com o molde são retornadas. Assim, fica claro que a chave calculada para as tuplas não tem o papel de identificador, mas sim de estabelecer um local inequívoco para a tupla e permitir a busca eficiente. Como espaços de tuplas são baseados em busca por conteúdo, nenhuma semântica é perdida pela existência de colisões.

Para encontrar uma tupla a partir de um molde $\bar{t}$, procede-se de maneira similar ao procedimento de inserção. Primeiro o molde é normalizado para $D+1$ dimensões, depois é calculado o $hash$ de cada campo, porém em vez de calcular uma única chave com a função $Hilbert$, é necessário encontrar o conjunto de todas as chaves na curva unidimensional que correspondem às possíveis tuplas que casam com $\bar{t}$. Esse procedimento é representado pela função $TemplateKeys(\bar{t})$. Por não ser o escopo principal deste trabalho, não abordamos em detalhes o procedimento de computação das chaves usando a curva de Hilbert. Mais informações podem ser obtidas a partir de outros trabalhos da literatura que também utilizam o mesmo mecanismo de indexação para realizar buscas multidimensionais e por intervalos [Li and Parashar 2005, Lee et al. 2005, Shen et al. 2008].

O estado de um nó é dado por duas estruturas locais:

• ts: é um espaço de tuplas local que provê os métodos $ts.out(t)$, $ts.rdp(\bar{t})$ e $ts.inp(\bar{t})$ com a semântica usual. Além disso, essa estrutura contém um atributo $ts.limits$, um intervalo tal que toda tupla t com $TupleKey(t) \notin ts.limits$ é descartada. A operação de união de dois espaços de tuplas locais ($ts' \cup ts''$), com intervalos de limite consecutivos, é dada pela união de todas as tuplas contidas nos dois operandos e pela união dos intervalos $limits$. $limits$ corresponde ao intervalo de chaves do segmento definido na segmentação;

• $pending$: é um conjunto com entradas da forma $\langle C_i, \bar{t} \rangle$ que indicam que o nó i está aguardando uma tupla que case com o molde $\bar{t}$.

Para que o algoritmo de replicação garanta a consistência, é necessário que ts seja idêntico em todas as réplicas e retorne o mesmo resultado quando invocado com os mesmos parâmetros. Isso não é garantido pela semântica básica de um espaço de tuplas e implementações diferentes podem retornar resultados diferentes. Para evitar esse problema e garantir o determinismo das réplicas, pode-se assumir o uso de uma única implementação por todos os nós corretos do sistema ou pode-se aumentar a semântica do espaço de tuplas local com propriedades de ordenação sobre as tuplas. O importante é assegurar que, dado que i e j são réplicas do mesmo segmento, $ts_i.inp(\bar{t}) = ts_j.inp(\bar{t})$, mesmo se mais de uma tupla case com $\bar{t}$.

Algoritmo 1 Obtenção e alteração do estado local do Espaço de Tuplas

```
1:  upon SegGetAppState(start, end) do /* Upcall da segmentação para obter estado */
2:      T ← {t ∈ ts : TupleKey(t) ∈ [start, end)} /* Tuplas com chave dentro do intervalo */
3:      P ← {⟨C_i, t̄⟩ ∈ pending : TemplateKeys(t̄) ∪ [start, end) ≠ ∅} /* Pendencias que se sobrepõem ao intervalo */
4:      return ⟨T, P⟩
5:  upon SegSetAppState(start, end, ⟨⟨T^1, P^1⟩, ..., ⟨T^n, P^n⟩⟩) do /* Upcall da segmentação para alterar estado */
6:      ts.limits ← [start, end) /* Altera limites do espaço de tuplas local */
7:      ts ← ⋃_{i=1}^{n} T_i /* Guarda união das tuplas dos subestados */
8:      pending ← ⋃_{i=1}^{n} P_i /* Guarda união das pendências dos subestados */
```

O Algoritmo 1 descreve como o espaço de tuplas responde às chamadas da camada de segmentação para obter e alterar o estado. Na chamada $SegGetAppState$ são retornadas as tuplas dentro do intervalo especificado (linha 2) e as buscas pendentes nas quais as chaves do molde tem intersecção com o intervalo (linha 3). Na chamada $SegSetAppState$ são recebidos um intervalo de chaves e uma lista de estados parciais que precisam ser unidos. Os limites de ts são alterados para o intervalo passado (linha 6), as tuplas de todos os estados parciais são unidas e armazenadas em ts (linha 7) e as buscas pendentes também são reunidas (linha 8).

5.1. Inserção de Tupla

Algoritmo 2 Inserção de tuplas (out)

```
1:  operation out(t) /* Código do cliente p */
2:      k ← TupleKey(t) /* Gera chave da tupla */
3:      nonce ← GenerateNonce() /* Gera nonce para requisição */
4:      resp ←⊥ /* Busca e invoca segmento responsável pela chave da tupla */
5:      while resp =⊥ do
6:          SegFind(k, k)
7:          wait for SegFindOk(S_q)
8:          resp ← SegRequest(S_q, ⟨OUT, nonce, t⟩)
9:      upon SegDeliver(C_p, ⟨OUT, nonce, t⟩) do /* Código do servidor q */
10:         if TupleKey(t) ∈ ts.limits then /* Calcula chave da tupla e verifica se está nos limites do segmento */
11:             ts.out(t) /* Insere tupla no espaço local */
12:             SegResponse(C_p, ⟨OUT_OK⟩) /* Responde para cliente */
13:             for all ⟨C_i, t̄⟩ ∈ pending : t ≤_m t̄ do /* Notifica clientes pendentes interessados na nova tupla */
14:                 SegNotify(C_i, ⟨READ_OK, t⟩)
15:                 pending ← pending \ {⟨C_i, t̄⟩}
```

A inserção de tupla, dada pela operação $out(t)$ (Alg. 2), consiste inicialmente em calcular a chave relacionada à tupla t a ser inserida, por meio da função $TupleKey$ executada no cliente (linha 2). Na sequência, um laço de repetição é iniciado para buscar o segmento responsável pela tupla (linhas 6 e 7) e invocar o segmento passando a requisição para inserir a tupla (linha 8). Essa busca e inserção devem ser repetidas até que a resposta correta seja recebida. O uso de um **nonce** garante que a mesma requisição não é executada mais de uma vez.

No servidor, ao receber uma requisição de inserção (linha 9), a chave da tupla é verificada (linha 10) e caso a tupla pertença realmente ao segmento esta é inserida no

espaço de tuplas ts local (linha 11). Na sequência, o nó procura alguma busca pendente $\langle C_i, \bar{t} \rangle$ na qual a tupla inserida case com o molde $\bar{t}$ (linha 13). Para cada caso em que a tupla casar com o molde, uma notificação é enviada ao nó que efetuou a busca em questão (linha 14) e a pendência é removida (linha 15). O cliente notificado pode, então, prosseguir com a leitura ou exclusão. Note que a exclusão em si não é executada nesse ponto, ou seja, um nó registrado ao tentar excluir uma tupla, após receber a notificação, precisa ainda realizar uma requisição de exclusão. Mais detalhes sobre a exclusão de tuplas serão apresentados na Seção 5.4.

5.2. Busca de Tupla

As operações de leitura e exclusão de tuplas apresentam uma característica similar: calculam as chaves relacionadas ao molde de busca e varrem todos os segmentos responsáveis por essas chaves a fim de descobrir quais segmentos possuem tuplas que casem com um molde. Após encontrar os segmentos relevantes, as operações de leitura simplesmente retornam as tuplas encontradas, enquanto as operações de exclusão invocam o segmento para remover a tupla. Dada essa similaridade, decidimos escrever um procedimento genérico, que é utilizado pelas operações de leitura e exclusão, e que realiza a busca nos segmentos e notifica as tuplas que forem encontradas. Esse procedimento foi idealizado para ser executado em uma **thread** separada para que a busca de tuplas possa ocorrer de maneira concorrente.

A busca de uma tupla a partir de um molde consiste em calcular as chaves do molde, buscar todos os segmentos responsáveis por chaves contidas no conjunto de chaves e consultar os segmentos para ler uma tupla que case com o molde buscado. Se uma tupla adequada for encontrada durante a consulta, esta é notificada para a **thread** principal da operação. Por outro lado, se nenhuma tupla for encontrada, no caso bloqueante, o algoritmo aguarda o recebimento de notificações de segmentos sempre que uma tupla que casa com o molde for encontrada; no caso não bloqueante a busca termina imediatamente. No lado do servidor, ao receber um pedido de leitura, o nó busca seu espaço de tuplas local usando rdp e retorna o resultado, seja este uma tupla ou $\perp$, para o cliente. Se a leitura for bloqueante e não houver tupla adequada (resultado $\perp$), então o servidor registra o pedido na lista de pendências para que possa notificar o cliente posteriormente.

Note que a busca de tupla é similar seja bloqueante ou não, portanto decidimos fazer um algoritmo geral $FindTuple$ (Alg. 3) que engloba as duas funcionalidades e recebe um parâmetro $block$ que indica o caso específico. As operações específicas simplesmente invocam $FindTuple$ passando o molde e o parâmetro $block$ adequado. Cada tupla t encontrada no segmento S é notificada com a chamada $FindTupleOk(S, t)$. Essa notificação é tratada de maneira diferente pelas operações de leitura e de exclusão.

Há um ponto que adiciona complexidade ao algoritmo da busca de tuplas: a possibilidade de uma consulta a um segmento não ser executada efetivamente e retornar $\perp$ (i.e. segmento reconfigurou após ser descoberto). Para lidar com esses casos, é necessário construir o algoritmo com várias "passadas" pelas chaves remanescentes, cujos segmentos responsáveis ainda não puderam ser consultados, até que todo o conjunto seja efetivamente coberto. O tratamento do estado durante as reconfigurações (Alg. 1) garante que se um segmento com intervalo de chaves é consultado, o mesmo intervalo não precisa ser buscado novamente, mesmo que o segmento sofra reconfiguração. Tuplas e nós pendentes são propagados para segmentos novos garantindo a semântica das operações.

O procedimento $FindTuple$, no cliente (Alg. 3), inicia pelo cálculo do conjunto de chaves associado ao molde da busca (linha 2). Depois, o algoritmo entra em um **loop** (linhas 3 a 16) até que todas as chaves sejam efetivamente buscadas. Na primeira iteração, as chaves buscadas são exatamente as chaves do molde recém calculadas. À medida que segmentos forem consultados, o conjunto de chaves remanescentes é reduzido até que não reste nenhuma chave a ser buscada e o **loop** termina. No interior do **loop**, dois conjuntos são criados para controlar as chaves que foram buscadas e os segmentos consultados na passada atual, respectivamente $foundKeys$ (linha 4) e $doneKeys$ (linha 5). Na sequência, o conjunto de chaves remanescente é dividido em intervalos contíguos e para cada um destes, $SegFind$ é invocado a fim de encontrar os segmentos responsáveis (linha 7).

Algoritmo 3 Busca de tuplas ($FindTuple$)

```
 1: procedure FindTuple(t̄, block) /* Código do cliente p */
 2:     remainingKeys ← TemplateKeys(t̄) /* Calcula intervalos de chaves do molde */
 3:     while remainingKeys ≠ ∅ do /* Repete até que todas as chaves do molde sejam cobertas */
 4:         foundKeys ← ∅ /* Chaves que tiveram segmento encontrado */
 5:         doneKeys ← ∅ /* Chaves que tiveram segmento consultado com sucesso */
 6:         for all [k, k'] ∈ remainingKeys do /* Busca segmentos de todas as chaves pendentes */
 7:             SegFind(k, k')
 8:         while remainingKeys ⊈ foundKeys do /* Repete até todas as chaves pendentes terem segmento encontrado */
 9:             wait for SegFindOk(S_i) /* Segmento encontrado */
10:             foundKeys ← foundKeys ∪ K(S_i) /* Marca chaves do segmento encontrado */
11:             resp ← SegRequest(S_i, ⟨READ, t̄, block⟩) /* Consulta segmento */
12:             if resp = ⟨READ_OK, t⟩ then
13:                 doneKeys ← doneKeys ∪ K(S_i) /* Se consultou com sucesso, marca chaves do segmento */
14:                 if t ≠⊥ then /* Se encontrou tupla, realiza notificação */
15:                     FindTupleOk(S_i, t)
16:         remainingKeys ← remainingKeys \ doneKeys /* Não achou tupla na passada, remove chaves consultadas */
17:     if block then /* Todas as chaves do molde foram consultadas e nenhuma tupla encontrada. */
18:         wait for SegNotifyDeliver(S_i, ⟨READ_OK, t⟩) /* Busca bloqueante, aguarda notificação com tupla */
19:         FindTupleOk(S_i, t) /* Retorna tupla notificada */
20:     else
21:         return /* Busca não-bloqueante, termina sem aguardar notificação */
22: upon SegDeliver(C_p, ⟨READ, t̄, block⟩) do /* Código do servidor q */
23:     t ← ts.rdp(t̄) /* Busca tupla localmente */
24:     if block ∧ t =⊥ then /* Verifica necessidade de registrar pendência */
25:         pending ← pending ∪ {⟨C_p, t̄⟩} /* Cliente marcado como pendente */
26:     SegResponse(C_p, ⟨READ_OK, t⟩) /* Envia resposta contendo tupla ou ⊥ */
```

Depois de realizar $SegFind$ em todas as chaves, um outro laço de repetição (linhas 8 a 15) realiza o recebimento dos certificados de segmentos buscados na etapa anterior e a consulta dos segmentos correspondentes. Para cada segmento encontrado (linha 9), as chaves correspondentes são marcadas como encontradas (linha 10) e o segmento é invocado para buscar tuplas que casem com o molde (linha 11). Além do molde, a requisição contém também o parâmetro $block$, indicando se a busca é bloqueante ou não.

Depois de invocado o segmento, a resposta recebida é testada para saber se a invocação foi realizada de fato (linha 12). Se a resposta foi efetivamente recebida, as chaves do segmento são marcadas como consultadas (linha 13) e o valor recebido é testado (linha 14) para saber se uma tupla que casa com o molde foi encontrada. Se for o caso, o algoritmo notifica a **thread** principal sobre a tupla encontrada.

Depois que todos os segmentos foram invocados o conjunto $foundKeys$ passa a conter todas as chaves remanescentes e o laço interno termina. Então, todas as chaves de segmentos que foram efetivamente consultados são removidas das chaves remanescentes (linha 16). Se todas as chaves foram consultadas, então o laço de repetição externo termina. A partir desse ponto, se $block = true$, então os segmentos consultados terão sido

informados que o cliente deseja ser registrado como pendente (linha 11), o cliente aguarda alguma notificação vinda de um segmento e repassa qualquer tupla informada (linhas 18 - 19); se $block = false$, então o algoritmo termina.

No lado do servidor, os nós que recebem uma requisição de leitura realizam uma busca no ET local (linha 23). Caso a tupla não exista e o cliente esteja realizando uma busca bloqueante, o mesmo é registrado nas pendências (linha 25) para ser notificado quando uma tupla adequada for inserida, conforme a operação out (Alg. 2, linha 14). Por fim, a requisição é respondida com o resultado da busca local (linha 26).

5.3. Leitura de Tupla

As operações rd e rdp (Alg. 4) utilizam $FindTuple$ (Alg. 3) e retornam a primeira tupla encontrada. Assim, o código consiste em iniciar a **thread** de busca, aguardar uma notificação e retornar a tupla encontrada. Em todo caso, a operação termina ou quando uma tupla é retornada, ou quando o procedimento de busca termina. As condições de término do procedimento de busca são descritas na Seção 5.2.

Algoritmo 4 Leitura de tuplas bloqueante (rd) e não bloqueante (rdp)

```
1: operation rd(t̄) /* Código do cliente */
2:     FindTuple(t̄, TRUE) /* Inicia procedimento de busca de tupla */
3:     wait for FindTupleOk(S_i, t) /* Aguarda notificação de tupla encontrada */
4:     return t /* Retorna a tupla */
5: operation rdp(t̄) /* Código do cliente */
6:     FindTuple(t̄, FALSE) /* Inicia procedimento de busca de tupla */
7:     upon FindTupleOk(S_i, t) do /* Recebe notificação de tupla encontrada ou término da busca */
8:         return t /* Retorna a tupla */
```

5.4. Exclusão de Tupla

Na exclusão de tupla (in e inp), o espaço de chaves que corresponde ao molde é buscado para encontrar tuplas que casem. Porém, ao encontrar uma tupla, uma nova invocação é realizada para excluir a mesma. Se outro cliente remover a mesma tupla antes, então essa tentativa de exclusão falha. Separar a operação em duas fases, busca e exclusão, permite que executar a busca em paralelo para intervalos de chaves distintos. Já as invocações para exclusão são realizadas em série, uma vez que somente uma tupla deve ser excluída em cada chamada de in ou inp. Além disso, caso uma requisição de exclusão falhe (resposta $\perp$), esta precisa ser repetida até que um resultado seja obtido indicando se a tupla foi ou não removida, pois se a requisição foi executada e a resposta simplesmente atrasou, a operação deve parar e retornar a tupla já removida, evitando a remoção de outra.

As operações de exclusão bloqueante e não bloqueante são similares. Uma diferença é com relação ao tipo de busca, i.e., o parâmetro $block$ do procedimento $FindTuple$, que é $true$ para in e $false$ para inp. A outra diferença é com relação às tentativas de remover uma tupla encontrada: na operação bloqueante, quando o cliente tenta remover uma tupla que não está mais presente no espaço de tuplas, este deve ser registrado como pendente no segmento; na operação não bloqueante isso não é necessário. Sendo pequenas as diferenças, construímos um procedimento único $RemoveTuple$ que recebe um parâmetro $block$, como em $FindTuple$ (Seção 5.2). As operações in e inp (Alg. 6) simplesmente chamam $RemoveTuple$ passando o molde e o valor do parâmetro $block$ adequado.

O procedimento $RemoveTuple$ (Alg. 5) incia chamando $FindTuple$, passando o molde e o parâmetro $block$ (linha 2). Na sequência, ocorre o tratamento das tuplas que são encontradas durante a busca (linha 3). Primeiro um $nonce$ é gerado (linha 4) a fim de evitar remover mais de uma vez a mesma tupla e garantir idempotência. Depois, o segmento

responsável é invocado para remover a tupla (linha 5). Essa requisição inclui a tupla e o parâmetro $block$. Se a requisição falhar, um **loop** busca novamente o segmento responsável pela tupla (linha 7) e envia novamente a requisição de exclusão para o segmento encontrado (linha 9), garantindo que uma resposta efetiva seja recebida. Se essa resposta contiver a tupla e não o valor $\perp$, então a tupla é retornada e o procedimento termina.

Algoritmo 5 Exclusão de tuplas ($RemoveTuple$)

```
1:  procedure RemoveTuple(t̄, block) /* Código do cliente p */
2:      FindTuple(t̄, block) /* Inicia busca de tuplas. Quando (se) a busca termina, RemoveTuple também termina */
3:      upon FindTupleOk(S_i, t) do /* Tupla encontrada */
4:          nonce ← GenerateNonce() /* nonce evita que mesmo molde remova mais de uma tupla */
5:          resp ← SegRequest(S_i, ⟨REMOVE, t, block, nonce⟩) /* Envia requisição para segmento remover tupla */
6:          while resp = ⊥ do /* Repete até que resposta seja recebida */
7:              SegFind(TupleKey(t)) /* Busca novamente segmento responsável pela tupla */
8:              wait for SegFindOk(S_j)
9:              resp ← SegRequest(S_j, ⟨REMOVE, t, block, nonce⟩) /* Repete invocação com mesmo nonce */
10:         if resp = ⟨REMOVE_OK, t⟩ ∧ t ≠ ⊥ then
11:             return t /* Retorna tupla que foi removida */
12: upon SegDeliver(C_p, ⟨REMOVE, t, block, nonce⟩) do /* Código do servidor q */
13:     t ← ts.inp(t) /* Busca e exclui tupla localmente */
14:     if block ∧ t = ⊥ then /* Verifica necessidade de registrar pendência */
15:         pending ← pending ∪ {⟨C_p, t̄⟩} /* Cliente marcado como pendente */
16:     SegResponse(C_p, ⟨REMOVE_OK, t⟩) /* Envia resposta contendo a tupla ou ⊥ */
```

No lado servidor, os nós do segmento invocado para excluir uma tupla agem de maneira similar ao caso da busca de tupla. A diferença é que inp, em vez de rdp, é chamado no ET local, e uma tupla específica é buscada, em vez de um molde. Assim como na busca, se a tupla não é encontrada e se $block = true$, o cliente é registrado como pendente.

Algoritmo 6 Exclusão de tuplas bloqueante (in) e não bloqueante (inp)

```
1:  operation in(t̄) /* Código do cliente p */
2:      return RemoveTuple(t̄, TRUE) /* Inicia procedimento de remover tuplas bloqueante */
3:  operation inp(t̄) /* Código do cliente p */
4:      return RemoveTuple(t̄, FALSE) /* Inicia procedimento de remover tuplas não bloqueante */
```

6. Discussão sobre o Espaço de Tuplas

Os algoritmos apresentados na seção anterior implementam um ET tolerante a intrusões sobre um **overlay** P2P. A operação out (Seção 5.1) consiste simplesmente em calcular a chave da tupla usando a curva de Hilbert, depois buscar e invocar o segmento responsável pela chave calculada. Pelas propriedades de $SegFind$ e $SegRequest$, a busca e invocação podem precisar ser repetidas caso o segmento buscado reconfigure antes da invocação ou caso a resposta demore a chegar no nó cliente. Para o segundo caso, o algoritmo utiliza um **nonce** que evita a repetição da inserção, o que violaria a semântica da operação out.

As operações rd e rdp (Seção 5.3) consistem em uma varredura do conjunto de chaves representado pelo molde buscado, de forma que cada segmento responsável por chaves deste conjunto é consultado sobre a existência de uma tupla adequada. De acordo com as propriedades das curvas de preenchimento de espaço, quanto mais geral é o molde utilizado, maior é o conjunto de chaves coberto e maior é o número de segmentos consultados. Buscas por moldes mais gerais, portanto, implicam em um maior número de trocas de mensagens, porém a consulta a diferentes segmentos é realizada em paralelo, reduzindo o tempo de resposta geral das operações rd e rdp. Assim como na inserção de tupla, é preciso repetir a busca e consulta a segmentos sempre que o segmento reconfigurar antes da invocação. No caso de rd (bloqueante), caso a tupla não seja encontrada o cliente aguarda notificações vindas dos segmentos consultados.

As operações in e inp (Seção 5.4) iniciam de maneira similar às operações de leitura. No caso de in (bloqueante), a operação também aguarda o recebimento de notificações posteriores caso a tupla não seja encontrada. Caso mais de uma tupla seja encontrada concomitantemente, é importante que as invocações de exclusão sejam realizadas em série, evitando que uma única operação in ou inp termine na exclusão de mais de uma tupla. Como a exclusão da tupla é uma invocação destrutiva, então um **nonce** também é utilizado pelo mesmo motivo, como na inserção.

A operação out realiza a verificação das buscas pendentes e notifica todos os nós bloqueados com moldes que casem com a tupla inserida. Todo nó notificado é excluído do conjunto de pendências. Para todos os nós bloqueados em operações rd, a simples notificação de uma tupla basta para encerrar o bloqueio. Por outro lado, no caso de nós bloqueados em operações in, o nó precisa ainda tentar remover a tupla para terminar a operação e caso a mesma tupla acabe sendo excluída antes, o nó precisa continuar bloqueado. Por esse motivo, a invocação para excluir a tupla pode fazer com que o nó entre novamente no conjunto de pendências. Uma característica interessante desse mecanismo de notificação é que se múltiplos nós bloqueados aceitam uma mesma tupla, a finalização de todas as leituras é garantida, mesmo que haja uma ou mais exclusões pendentes.

7. Trabalhos Relacionados

Linda [Gelernter 1985] é uma linguagem de coordenação que introduziu o conceito de ET e comunicação generativa. Diversas implementações de ET foram desenvolvidas e incluídas em plataformas de coordenação, como JavaSpaces (`http://www.jini.org/wiki/JavaSpaces\_Specification`) e IBM TSpaces (`http://www.almaden.ibm.com/cs/TSpaces/`). Essas soluções armazenam as tuplas em um servidor centralizado, limitando a escalabilidade e a tolerância a falhas.

Algumas soluções usam replicação para garantir disponibilidade e confiabilidade do ET [Xu and Liskov 1989, Hansen and Cannon 1994, Bakken and Schlichting 1995, Bessani et al. 2008]. O DepSpace [Bessani et al. 2008] é um ET tolerante a faltas bizantinas que possui propriedades de segurança como confidencialidade e controle de acesso.

Lime [Picco et al. 1999] é um ET para redes móveis, onde cada nó possui seu ET local. Uma visão distribuída do ET é construída a partir do agrupamento do ET local com o ET de nós vizinhos na rede. Apesar da distribuição, o sistema não tolera saídas ou falhas, uma vez que as tuplas locais não são replicadas.

Comet [Li and Parashar 2005] é uma implementação de espaços de tuplas sobre uma DHT **Chord** que utiliza curvas de Hilbert para distribuir as tuplas pelos nós do **overlay**. No entanto, o mesmo não utiliza replicação e não tolera faltas.

8. Conclusão

Este artigo apresentou uma construção de espaço de tuplas sobre a infraestrutura de segmentação proposta em [Böger et al. 2012]. O espaço de tuplas foi elaborado na forma de algoritmos distribuídos que realizam as operações seguindo a semântica usual. Uma análise informal dos algoritmos levantou aspectos específicos da implementação, incluindo limitações e formas de contorná-las. Além de servir como demonstração da utilidade da segmentação, os algoritmos representam uma contribuição por si mesmos, uma vez que, conforme apresentado nos trabalhos relacionados, há poucas iniciativas para usar espaços de tuplas em P2P e nenhum trabalho nesse contexto somando-se tolerância a intrusões.

Referências

Bakken, D. and Schlichting, R. (1995). Supporting fault-tolerant parallel programming in linda. *Par. and Dist. Syst., IEEE Trans. on*, 6(3):287 –302.

Baldoni, R., Querzoni, L., Virgillito, A., Jimenez-Peris, R., and Virgillito, A. (2005). Dynamic quorums for dht-based p2p networks. In *Net. Comp. and App., 4th IEEE Int. Symp. on*, pages 91–100.

Bessani, A. N., Alchieri, E. P., Correia, M., and Fraga, J. S. (2008). Depspace: a byzantine fault-tolerant coordination service. *SIGOPS Oper. Syst. Rev.*, 42(4):163–176.

Böger, D. S., Fraga, J., Alchieri, E., and Wangham, M. (2012). Intrusion-tolerant shared memory through a p2p overlay segmentation. In *Adv. Inf. Net. and App. (AINA), 2012 IEEE 26th Int. Conf. on*, pages 779 –786, Fukuoka, JP. IEEE.

Castro, M., Druschel, P., Ganesh, A., Rowstron, A., and Wallach, D. (2002). Secure routing for structured peer-to-peer overlay networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):299–314.

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

Gelernter, D. (1985). Generative communication in linda. *ACM Trans. Prog. Lang. Syst.*, 7(1):80–112.

Hansen, R. and Cannon, S. (1994). An efficient fault-tolerant tuple space. In *Fault-Tol. Par. and Dist. Syst., 1994., Proc. of IEEE Work. on*, pages 220 –225.

Lamport, L., Malkhi, D., and Zhou, L. (2010). Reconfiguring a state machine. *SIGACT News*, 41(1):63–73.

Lamport, L., Shostak, R., and Pease, M. (1982). The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401.

Lawder, J. and King, P. (2000). Using space-filling curves for multi-dimensional indexing. In Lings, B. and Jeffery, K., editors, *Adv. in Dat.*, volume 1832 of *LNCS*, pages 20–35. Springer Berlin / Heidelberg.

Lee, J., Lee, H., Kang, S., Choe, S., and Song, J. (2005). Ciss: An efficient object clustering framework for dht-based peer-to-peer applications. In Ng, W., Ooi, B.-C., Ouksel, A., and Sartori, C., editors, *Dat., Inf. Syst., and Peer-to-Peer Comp.*, volume 3367 of *LNCS*, pages 215–229. Springer Berlin Heidelberg.

Li, Z. and Parashar, M. (2005). Comet: a scalable coordination space for decentralized distributed environments. In *Hot Topics in Peer-to-Peer Systems, 2005. HOT-P2P 2005. 2nd Int. Work. on*, pages 104 – 111.

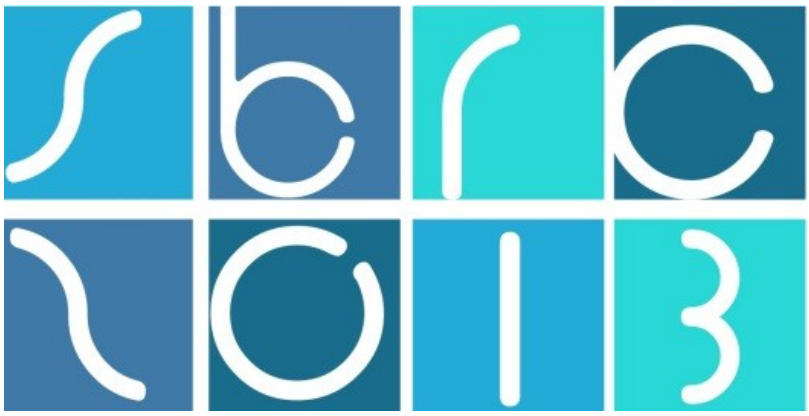
Picco, G., Murphy, A., and Roman, G.-C. (1999). Lime: Linda meets mobility. In *Soft. Eng., 1999. Proc. of the 1999 Int. Conf. on*, pages 368 –377.

Rowstron, A. and Druschel, P. (2001). Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In Guerraoui, R., editor, *Middleware 2001*, volume 2218 of *LNCS*, pages 329–350. Springer, Berlin.

Shen, D., Shao, Y., Nie, T., Kou, Y., Wang, Z., and Yu, G. (2008). Hilbertchord: A p2p framework for service resources management. In Wu, S., Yang, L., and Xu, T., editors, *Adv. in Grid and Perv. Comp.*, volume 5036 of *LNCS*, pages 331–342. Springer Berlin Heidelberg.

Wallach, D. (2003). A survey of peer-to-peer security issues. In Okada, M., Pierce, B., Scedrov, A., Tokuda, H., and Yonezawa, A., editors, *Soft. Sec. - Theo. and Syst.*, volume 2609 of *LNCS*, pages 253–258. Springer Berlin / Heidelberg.

Xu, A. and Liskov, B. (1989). A design for a fault-tolerant, distributed implementation of linda. In *Fault-Tol. Comp., 19th Int. Symp. on*, pages 199 –206.



31º Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos
Brasília-DF

XIV Workshop de Testes e Tolerância a Falhas

Sessão Técnica 3

Escalonamento e Virtualização

Escalonamento Tolerante a Falhas para Clusters Multicores

Brevik Ferreira da Silva, Wellison Moura dos Santos, Idalmis Milián Sardiña, Livia de Mesquita Teixeira, Felipe de Albuquerque

[1]Escola de Ciência e Tecnologia – Universidade Federal do Rio Grande do Norte (UFRN)
Av. Sen. Salgado Filho, 3000 - Lagoa Nova Natal - RN, 59078-970 – Natal – RN – Brazil

{idalmismilian}@ect.ufrn.br

Abstract. *Large-scale parallel applications running with increased performance and fault-free is a challenge of the high performance computing. However, for best results is important the efficient use of available resources, exploring for example the shared and distributed memories of the new multicores architectures. This paper proposes a hybrid approach for fault-tolerant scheduling on clusters based on multicores processors. A case study is proposed for a parallel application modeled by a Directed Acyclic Graph (GAD) using the hybrid programming OpenMP and MPI. The proposal should discuss the advantages that this model can bring to these architectures, compared with the previous approach.*

Resumo. *Um dos principais desafios da computação de alto desempenho é executar aplicações paralelas de grande porte com maior desempenho e livre de falhas. Entretanto, para obter melhores resultados é primordial o aproveitamento eficiente dos recursos disponíveis, explorando por exemplo o uso das diferentes memórias compartilhadas e distribuídas em novas arquiteturas multicores. Este trabalho propõe uma abordagem híbrida para o escalonamento tolerante a falhas sobre clusters baseados em processadores multicores. Um estudo de caso é proposto para uma aplicação paralela modelada por um Grafo Acíclico Direcionado (GAD) usando programação paralela híbrida OpenMP e MPI. O estudo proposto deve analisar as vantagens que este modelo pode trazer para estas arquiteturas, comparado com a abordagem anterior.*

1. Introdução

Clusters multicore tem se tornado uma plataforma popular em computação paralela ou de alto desempenho. Na lista publicada no Top500 de supercomputadores, a maioria das máquinas representam arquiteturas de *clusters* com processadores multicores. Intuitivamente, os processadores multicores podem dividir a carga de trabalho distribuindo as tarefas nos diferentes cores ou núcleos de processamento. Mesmo assim, comparados com *clusters* SMP ou NUMA, aplicações sobre *clusters* multi-cores não mostram sempre ótimos desempenhos nem escalabilidade. Atualmente, é fundamental um estudo maior das características destas arquiteturas e seus efeitos sobre o comportamento das aplicações que executam neles. Duas estratégias importantes empregadas nos atuais processadores multicores estão nos processadores da Intel e da AMD. Os da Intel provêm uma *cache* L2 compartilhada que igual a AMD emprega enlaces *HyperTransport* para rápidas transferências de dados. Assim, estas arquiteturas apresentam eficientes protocolos e ambas executam compartilhamentos rápidos de dados através dos cores. Entretanto,

diversos estudos são realizados para explorar as hierarquias de memórias, assim como analisar os efeitos e benefícios da execução das aplicações neles. Para muitas aplicações científicas, o custo da troca de mensagens sobre as redes e *clusters* de memória distribuída já é bastante conhecido, mas a análise de custo e efeitos explorando os múltiplos núcleos ou cores exige novas pesquisas.

Desenvolvedores de aplicações científicas e técnicas em geral, necessitam paralelizar altos volumes de código garantindo eficiência e portabilidade. O uso de arquiteturas multiprocessadas com memória compartilhada tem criado uma demanda urgente, precisando de uma forma diferente de programar as aplicações. A ferramenta OpenMP [Ope 2008], por exemplo, foi desenvolvida para direcionar estes aspectos e criar um padrão para a programação dos multiprocessadores. OpenMP consiste de um conjunto de diretivas de compilação e bibliotecas estendidas em Fortran, C, C++ para expressar o paralelismo da memória compartilhada. A programação paralela em OpenMP é bastante atual representando um padrão hoje. Por outro lado, a era da programação paralela marca a popularidade dos *softwares* MPI [MPI 2009] e Open MP como programação híbrida e o emprego de *clusters* multicores como as plataformas de *hardware* em inúmeras organizações e lares. Desta forma, surge uma necessidade eminente de estudantes e profissionais nestas áreas aprendam novas metodologias que combinem as funções tradicionais do padrão MPI com novas diretivas de OpenMP com o objetivo de obter melhores resultados.

Muito dos códigos híbridos usando MPI e OpenMP para executar as aplicações, são baseados em um modelo de estrutura hierárquica, que torna possível a exploração de ambas linguagens. O objetivo é tirar vantagens das melhores características de ambos os paradigmas de programação. A utilização da dupla MPI e OpenMP está emergindo como um padrão de fato [Rabenseifner et al. 2009b, Rabenseifner et al. 2009a, Su et al. 2004, Aversa et al. 2005, Chorley et al. 2009], ambos são dois padrões bem estabelecidos e possuem sólida documentação. No entanto, em cada caso ou aplicação deve ser analisado o modelo híbrido a ser utilizado, e sempre levando em consideração a arquitetura do *hardware*.

Por outro lado, para atingir alto desempenho ao executar aplicações paralelas de grande porte em MPI, diferentes trabalhos sobre escalonamento de tarefas, integram heurísticas de escalonamento com mecanismos tolerantes a falhas. Existem algoritmos de escalonamento que para tratar falhas empregam estratégias de replicação de tarefas baseadas no esquema primária-*backup* [Benoit et al. 2008, Qin and Jiang 2006, Al-Omari et al. 2005, Sardina et al. 2011a, Sardina et al. 2011b]. Neste caso, para a replicação ativa [Benoit et al. 2008, Anne Benoit and Robert 2008] ambas cópias da tarefa são executadas simultaneamente, o que pode sobrecarregar bastante o sistema distribuído. Já com a técnica de replicação passiva Qin2006, MilianBoeresDrummond2011], a *backup* de uma tarefa somente é ativada quando detectada falha na primária. A replicação passiva tem se mostrado na literatura uma alternativa interessante à replicação ativa, dado que não necessita de uso de recursos extras, com custos de execução de *backups* mais reduzidos.

Em [Sardina et al. 2011b] é proposto um algoritmo de escalonamento estático, inicialmente baseado em [Qin and Jiang 2006] ao empregar uma heurística do tipo *list scheduling* para escalonar as tarefas primárias e *backups*, com tolerância de uma falha *crash*. Esta abordagem adiciona o escalonamento bi-objetivo de [Sardina et al. 2011a] e acrescenta maior flexibilidade com a introdução de novos conceitos e critérios para o

escalonamento de *backups*.

Neste trabalho é estendida a pesquisa feita em [Sardina et al. 2011b] com o objetivo de explorar as novas arquiteturas multicores e propor novos estudos que apliquem as estratégias propostas de escalonamento e tolerância a falhas a estas arquiteturas. O artigo apresenta o estudo de caso de uma aplicação paralela baseada no algoritmo de escalonamento tolerante a falhas e que usa uma abordagem híbrida para a implementação paralela. Um dos desafios principais deste projeto é executar aplicações paralelas de problemas de grande porte com maior desempenho usando estratégias escalonamento de tarefas e tolerância a falhas que aproveitem o processamento multicore.

Inicialmente, a abordagem híbrida proposta para implementar o escalonamento tolerante a falhas é testada sobre um pequeno ambiente que permite executar programas paralelos. Depois os testes são realizados em outras arquiteturas maiores e mais complexas.

O artigo está organizado como segue. A seção 2 descreve o modelo da aplicação e da arquitetura empregados. A seção 3 mostra o algoritmo de escalonamento tolerante a falhas. A seção 4 explica a abordagem proposta neste trabalho para o escalonamento. Inicialmente, esta seção introduz conceitos de programação paralela híbrida e depois detalha um caso de estudo para um GAD de aplicação da Eliminação de Gauss. As últimas seções apresentam o ambiente de testes utilizado e discussões relacionadas.

2. Modelo do sistema

O trabalho proposto se aplica a um modelo de sistema que descreve características da aplicação e da arquitetura nas Seções 2.1 e 2.2, respectivamente. A modelagem parte da necessidade de representar o ambiente de trabalho a ser utilizado e deve suportar a execução de grandes e variadas aplicações paralelas.

2.1. Modelo da aplicação

Neste trabalho as aplicações são modeladas por *Grafos Acíclicos Direcionados* (GAD), com $G = (V, E, e, c)$, onde V é o conjunto de vértices (tarefas), E a relação de precedência, $e(v_i)$ com $v_i \in V$, o peso de execução associado à tarefa v_i e, $c(v_j, v_i)$ com $(v_j, v_i) \in E$, o peso de comunicação associado ao arco (v_j, v_i). Se $(v_j, v_i) \in E$ então, a execução de v_i não pode ser iniciada enquanto não seja completada a execução de v_j e os dados de v_j para v_i sejam recebidos por este. O conjunto de predecessores imediatos de v_i é denotado por $Pred(v_i)$, enquanto $Succ(v_j)$ são os sucessores imediatos de v_j. O GAD utilizado neste trabalho é uma paralelização do método matemático Eliminação de Gauss utilizado para solucionar sistemas de equações lineares. A estrutura deste grafo (Figura 1) possui vértices(tarefas) com pesos de computação variável, uma vez que os pesos das tarefas são maiores inicialmente na parte superior do GAD e diminuem a cada nível. Este GAD serve como modelo para estudar aplicações heterogêneas em relação às tarefas.

2.2. Modelo da arquitetura

Em relação a arquitetura, $P = \{p_0, p_1, ..., p_{m-1}\}$ é o conjunto de m processadores multicores, sendo que a cada p_j é associado o índice de retardo (*computational slowdown index*), denotado por $csi(p_j)$, sendo esta métrica inversamente proporcional ao poder computacional de p_j. O tempo de execução da tarefa v no processador p_j é dado

por $eh(v, p_j) = e(v) \times csi(p_j)$. Para duas tarefas adjacentes v_i e v_j alocadas em processadores distintos p_l e p_k, respectivamente, supõe que o custo associado à comunicação de $c(v_i, v_j)$ dados é definido como $ch(v_i, v_j) = c(v_i, v_j) \times L(p_l, p_k)$, onde a latência $L(p_l, p_k)$ é o tempo de transmissão por *byte* sobre o *link* (p_l, p_k). Cada processador p_j pode ter mais de um núcleo ou *core* n_i, e pode ser representado como $p_j = n_0, n_1, ..., n_{k-1}$.

São consideradas falhas permanentes de processador, eventos independentes entre si, e que ocorrem de acordo com uma distribuição de *Poisson* com probabilidade de falha $FP(p_j)\ \forall\ p_j\ \in\ P$ e valor constante, conforme [Qin and Jiang 2006]. $FP(p_j)$ representa a quantidade de falhas por unidade de tempo que podem ocorrer em p_j. O custo de confiabilidade $RC(v, p_j)$ de execução de v em p_j é definido como $RC(v, p_j)\ =\ FP(p_j) \times eh(v, p_j)$ e deve ser minimizado para aumentar a confiabilidade. Sendo $ltask(p_j)$ a lista de tarefas atribuídas a p_j, o custo de confiabilidade associado à p_j é $RC_p(p_j) = \sum_{\forall v \in ltask(p_j)} RC(v, p_j)$. Para um sistema P com m processadores, o custo de confiabilidade de escalonar uma aplicação pode ser definido como $RC(G, P)\ =\ \sum_{\forall p_j \in P} RC_p(p_j)$. Assim, a confiabilidade da aplicação G é dada por $R = e^{-RC(G,P)}$.

3. Algoritmo de Escalonamento Estático Tolerante a Falhas

O algoritmo de escalonamento estático proposto em [Sardina et al. 2011b] utiliza uma política do tipo *list scheduling* para escalonar as tarefas, incluindo mecanismos que permitem tolerar falhas permanentes de processador. Para isto, emprega uma técnica de replicacão passiva chamada primária-*backup* [Al-Omari et al. 2005, Qin and Jiang 2006]. Um dos objetivos deste algoritmo é minimizar o tempo total de execucção *(makespan*) de tal maneira que mais tarefas possam ser executadas. Outro objetivo é obter um sistema mais confiável reduzindo o custo de confiabilidade durante o escalonamento e permitir que falhas permanentes de processador possam ser toleradas. Portanto, a estratégia de escalonamento estático proposta em [Sardina et al. 2011b] apresenta os objetivos: encontrar uma alocação para as primárias das tarefas da aplicação sobre uma arquitetura proposta; de acordo com a alocação das primárias, encontrar uma alocação para as *backups* que permita tolerar falhas permanentes de processador; e minimizar o *makespan* e maximizar a confiabilidade da aplicação mesmo na presença de falhas. O algoritmo consiste das seguintes 3 etapas:

1. Ordenar tarefas usando um critério de prioridade: OrdenaTarefas();
   Na primeira etapa as tarefas são ordenadas por seus *b-levels* (*static bottom level*), conforme [Topcuouglu et al. 2002], denotado por $prior(v),\ \ \forall v \in V$. A lista de tarefas V_{ord} em ordem decrescente de $blevel(v)$, definido como:

$$blevel(v) = \begin{cases} \overline{et(v)} & \text{se } succ(v) = \emptyset \\ max_{u \in succ(v)} \left\{ \overline{et(v)} + \overline{ct(v,u)} + blevel(u) \right\} & \text{em outro caso} \end{cases}$$

   onde $\overline{et(v)}$ é o tempo de execução médio de v considerando todos os processadores e $\overline{ct(v,u)}$ é o tempo de comunicação médio, considerando todos os *links*. Em heurísticas *list scheduling* sobre ambientes heterogêneos, a escolha da tarefa usando *b-level* tem mostrado melhor desempenho para um número maior de GADs [Topcuouglu et al. 2002].

2. Escalonar cópias primárias: EscalonaPrimarias();
   Na segunda etapa, usando um *list scheduling* para escalonar as tarefas, a lista ordenada por $prior(v_i)$ é percorrida e, para cada tarefa da lista é efetuada uma procura pelo melhor processador de acordo com os seguintes critérios. Seja $v \in V$ a próxima tarefa a ser escalonada, o melhor processador $p_j \in P$ a ser escolhido para execução de v é aquele que minimiza a função de custo ponderada $f(v, p_j) = \alpha_1 EFT(v, p_j) + \alpha_2 RC(v, p_j)$. Ou seja, é escolhido o processador que satisfaz $F(v, p) = min_{\forall p_j \in P'} \{f(v, p_j)\}$. De acordo com a função $f(v, p_j)$, tarefas mais críticas em relação ao seu tempo de execução e comunicação tendem a ser escalonadas em processadores mais confiáveis e mais rápidos, aumentando o desempenho e a probabilidade da aplicação não falhar [Sardina et al. 2011b]. Dependendo do caso, uma tarefa v com granularidade mais fina pode ser alocada diretamente num core n_i daquele processador p_j permitindo que outras tarefas, por exemplo tarefas paralelas, sejam alocadas no mesmo processador mas em cores distintos. Novos critérios de escalonamento como este são utilizados para aproveitar o processamento multicore e reduzir comunicação, dependendo do caso.
   Para calcular o tempo de início $EST(v, p_j)$ da tarefa v no processador p_j é considerada uma política de inserção de tarefas em espaços ociosos de processador conforme [Topcuouglu et al. 2002]. O peso α_i, associado a cada critério, é uma variável $0 \leq \alpha_i \leq 1$ que representa o nível de importância de cada critério ($i = 1, 2$).
3. Escalonar cópias *backups*: EscalonaBackups(). Para obter o escalonamento das *backups* é utilizado também um *list scheduling* com a mesma função ponderada e critérios de escalonamento do passo anterior (2), junto a uma estratégia primária-*backup* como proposto em [Sardina et al. 2011b]. O escalonamento das tarefas *backups* (ver Figura 1) é realizado logo depois das primárias e verifica-se que seja satisfeito o conjunto de critérios propostos em [Sardina et al. 2011b].

As informações geradas pelo escalonamento final deste algoritmo são utilizadas posteriormente por uma ferramenta gerenciadora para executar a aplicação paralela e recuperar a aplicação em caso de falhas.

4. Abordagem Híbrida para o Escalonamento da Aplicação

A aplicação escalonada usando o algoritmo proposto, deve ser executada em um ambiente apropriado de computação paralela e com linguagens de programação específicas como C++ ou Fortran, que podem incluir as diretivas e funções necessárias para a programação paralela, por exemplo o padrão MPI. O artigo [Sardina et al. 2011a] mostra a implementação de uma ferramenta MPI ou *middleware* [Boeres and Rebello 2004, de P. Nascimento et al. 2005, Sardina 2010] (SGA) que foi modificado para gerenciar o escalonamento tolerante a falha das tarefas da seção anterior e executar a aplicação paralela em caso de falhas. Estas implementações em MPI não consideram as características multicore dos processadores. Este trabalho propõe uma nova abordagem híbrida para a implementação do escalonamento tolerante a falhas. Inicialmente, é feita nesta seção uma introdução do estilo de programação que pode ser utilizada e as razões que levam a esta escolha. Logo depois é apresentado um caso de estudo para uma aplicação específica utilizando a estratégia híbrida proposta.

4.1. Programação Híbrida: OpenMP e MPI

A biblioteca MPI (Message-Passing Interface) [MPI 2009] é uma especificação ou interface para troca de mensagens que representa um paradigma de programação paralela. Os dados são movidos de um espaço de endereçamento de um processo para o espaço de endereçamento de outro processo, através de operações específicas para intercâmbio de mensagens. As principais vantagens do estabelecimento de um padrão deste tipo são a portabilidade e a facilidade de utilização. Em um ambiente de comunicação de memória distribuída é vantajoso ter a possibilidade de utilizar rotinas em mais alto nível ou abstrair a necessidade de conhecimento e controle das rotinas de passagem de mensagens, como por exemplo *sockets*. Como a comunicação por troca de mensagens é padronizada por um fórum de especialistas, MPI tende oferecer eficiência, escalabilidade e portabilidade.

MPI além de ter como alvo de sua utilização as plataformas de *hardware* de memórias distribuídas, o mesmo pode ser utilizado também em plataformas de memória compartilhada como as arquiteturas SMPs e NUMA. E ainda, torna-se mais aplicável em plataformas híbridas, como cluster de máquinas NUMA. Todo o paralelismo em MPI é explícito, ou seja, de responsabilidade do programador, e implementado com a utilização dos construtores da biblioteca no código.

OpenMP [Ope 2008] é uma API (Application Program Interface) para programação em C/C++, Fortran entre outras linguagens, que oferece suporte para programação paralela em computadores que possuem uma arquitetura de memória compartilhada. O modelo de programação adotado pelo OpenMP é bastante portável e escalável, podendo ser utilizado numa gama de plataformas que variam desde um computador pessoal até supercomputadores. OpenMP é baseado em diretivas de compilação, rotinas de bibliotecas e variáveis de ambiente. O OpenMP provê um padrão suportado por quase todas as plataformas ou arquiteturas de memória compartilhada. Um detalhe interessante e importante é que isso é conseguido utilizando-se um conjunto simples de diretivas de programação. Em alguns casos, o paralelismo é implementado usando 3 ou 4 diretivas. É, portanto, correto afirmar que o OpenMP possibilita a paralelização de um programa sequencial de forma amigável. Como o OpenMP é baseado no paradigma de programação de memória compartilhada, o paralelismo consiste então de múltiplas *threads*.

Entretanto, OpenMP precisa de suporte para alocação distribuída de estruturas compartilhadas, podendo causar gargalos em alguns sistemas. Uma vez que o conjunto de dados é muito grande, há muitos *misses* na *cache*, afetando severamente o desempenho e a escalabilidade; otimizações nos códigos OpenMP não são simples, ou seja, demandam alto conhecimento do programador. Para minimizar esses problemas, trabalhos como [Su et al. 2004], por exemplo, utilizam recursos de alocação disponíveis na máquina da SGI nos testes. A implementação do OpenMP da SGI (Silicon Graphics, Inc) aceita parametrizações que afetam o modo como o sistema aloca os dados e como os dados são transferidos em tempo de execução para minimizar a latência de acesso aos dados na memória.

A programação híbrida com ambos paradigmas OpenMP e MPI é o resultado de mesclar a paralelização explícita de grandes tarefas em MPI, com a paralelização de tarefas mais simples em OpenMP [Rabenseifner et al. 2009b]. Em um alto nível, o programa fica hierarquicamente estruturado como uma série de tarefas MPI, cujo código sequencial está enriquecido com diretivas OpenMP para adicionar *multithreading* e aproveitar as

características da presença de memória compartilhada e multiprocessadores dos nós. O uso de OpenMP deve acrescentar *multithreading* aos tradicionais processos MPI, o que de certa maneira, deixa o desenvolvimento dos programas mais complexos.

Quando não há transmissão de mensagens dentro do nó, as otimizações do MPI não são necessárias. As partes do OpenMP devem ser otimizadas para a arquitetura do nó pelo programador, por exemplo, empregando cuidados com a localidade dos dados na memória em nós NUMA, ou usando mecanismos de afinidade. Portanto, a maneira mais simples e segura de combinar o MPI com o OpenMP é utilizar as directivas MPI apenas fora das regiões paralelas do OpenMP. Quando isso acontece, não há qualquer problema nas chamadas da biblioteca do MPI.

4.2. Abordagem proposta

Em trabalhos anteriores [Sardina et al. 2011a, Sardina et al. 2011b], uma ferramenta MPI gerenciadora recebe como entrada os arquivos com a informação gerada pelo algoritmo de escalonamento estático tolerante a falhas. Para cada processador, um arquivo mostra a maneira em que foram escalonadas as tarefas primárias e *backups*. O escalonamento das *backups*, descreve como será o comportamento da aplicação em caso de falha, ou seja, diferentes execuções possíveis da aplicação, dependendo do processador que falha.

Um mecanismo para deteção e recuperação de falhas forma parte da ferramenta híbrida, e redefine as principais funções MPI que a aplicação, modelada por um GAD, pode usar (MPI_Send e MPI_Recv). As modificações destas funções permite tratar as falhas em conjunto com processos gerentes que monitoram e processam a ocorrência de falhas no sistema. Para a aplicação tornar-se tolerante a falhas deve ser compilada com esta ferramenta e incluir os arquivos necessários com as informações geradas pelo escalonamento proposto para recuperar a aplicação. Portanto, para executar a aplicação, nenhum código é alterado e como resultado da compilação, a tolerância a falhas é automaticamente inserida no programa. Consequentemente, a aplicação compilada fica pronta para executar em paralelo e com a capacidade de recuperar-se em caso de falhas.

Inicialmente, foi implementada a detecção com o uso de *error handlers*, onde funções redefinidas de MPI (envio e recebimento) interceptam os erros que ocorrem durante a execução da aplicação. Assim, uma identificação de falhas pelo MPI é habilitado através de *error handlers* associados aos comunicadores. A cada comunicador é associado o *MPI ERRORS RETURN*, o qual permite que as funções retornem um código de erro na ocorrência de falhas. Outros mecanismos de tolerância a falhas são utilizados seguindo os trabalhos [Sardina 2010, da Silva 2010] para recuperar a aplicação. A seção 4.2.1 descreve um exemplo de como procede o mecanismo híbrido para executar e recuperar a aplicação paralela em caso de falha.

4.2.1. Caso de uso: Aplicação de Gauss

A Figura 1 mostra um exemplo do GAD da aplicação paralela Eliminação de Gauss para 9 tarefas. Esta aplicação está formada por dois tipos de tarefas em função da posição no grafo, denotadas por $T_{k,k}$ e $T_{k,j}$ respectivamente. De forma geral, podemos descrever o código híbrido (MPI e OpenMP) proposto para esta aplicação como mostrado

a seguir, onde $T_{k,k}$ representa as tarefas $V_0(T_{1,1})$, $V_4(T_{2,2})$ e $V_7(T_{3,3})$ na Figura 1 e $T_{k,j}$ são as outras tarefas restantes do GAD.

1. **para** $k = 1 \ldots m-1$ **faça (MPI)**
2. $T_{k,k}$: {
3. **recebe** $(col_k, T_{k-1,k})$
4. **para** $i = k+1 \ldots m$ **faça (OpenMP)**
5. $a_{ik} = a_{ik}/a_{kk}$
6. **envia** $(col_k, T_{k,j})$ }
7. **para** $j = k+1 \ldots m$ **faça (MPI)**
8. $T_{k,j}$: {
9. **recebe** $(col_k, T_{k,k})$
10. **para** $i = k+1 \ldots m$ **faça (OpenMP)**
11. $a_{ij} = a_{ij} - a_{ik} * a_{kj}$
12. **envia** $(col_j, T_{k+1,j})$ }

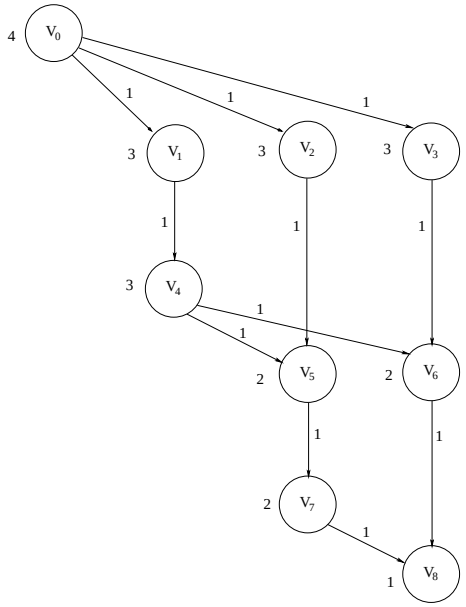


Figura 1. Eliminação de Gauss com 9 tarefas

Uma abordagem para explorar o *hardware* multicore disponível na arquitetura, mesmo na presença de falhas, é atribuir as tarefas T_{kk} e T_{kj} do GAD de Gauss (Figura 1) aos mesmos processadores p_j escolhidos pelo escalonamento estático tolerante a falhas da Seção 3. Dentro do *loop* de cada tarefa (T_{kk} ou T_{kj}) o código é paralelizado somente com OpenMP decompondo a tarefa em subtarefas menores. Para isto, é associada uma *thread* a cada core n_i de cada processador p. Na parte mais externa do código das tarefas do GAD T_{kk} e T_{kj}, a decomposição segue a distribuição do GAD, utilizando MPI para realizar a troca de mensagens entre as tarefas na forma que mostra a Figura 1. Observe que as tarefas T_{kj} recebem as mensagens das tarefas T_{kk}, ou seja, $V_1(T_{12})$, $V_2(T_{13})$ e $V_3(T_{14})$ na Figura 1, recebem da tarefa $V_0(T_{11})$ a coluna 1 da matriz calculada, usando as diretivas de comunicação de MPI. As tarefas T_{kk}, exceto a primeira, recebe mensagem da tarefa T_{k-1k} ($j = k$), assim neste caso $V_4(T_{22})$ recebe a coluna 2 de $V_1(T_{12})$.

Para o modelo da arquitetura proposto (Seção 2.2), o escalonamento pode também alocar uma tarefa considerando um core n_i no lugar de um processador p_j, em casos como este a comunicação MPI entre algumas tarefas pode ser desnecessária, a mensagem é lida pela mesma memória compartilhada reduzindo custos de comunicação. Neste caso de Gauss por exemplo, várias tarefas T_{kj} (primárias ou *backups*) de um mesmo nível k na Figura 1, podem ser alocadas em cores diferentes n_i de um mesmo processador p_j. Assim uma única mensagem MPI, comum a todas as tarefas T_{kj}, é recebida pelo processador p_j no lugar de j mensagens. Para isto, as tarefas T_{kj} são associadas a *thread*s com OpenMP. Estudos comparativos analisando a diferença de escalonar mais tarefas, por exemplo T_{kj} (não dependentes ou paralelas), dentro de um mesmo processador em cores diferentes, em vez destas mesmas tarefas em processadores separados são realizados. Isto permite comparar custos de comunicação usando memória distribuída contra custos de execução na memória compartilhada em função do tipo de escalonamento escolhido para cada tarefa. Novos critérios de escalonamento podem surgir desta pesquisa, assim como uma nova função ponderada que considere estes custos como objetivos ou parâmetros do escalonamento.

Desta forma, a execução paralela segue o escalonamento proposto na Seção 3 e em caso de falha de processador p_j, as *backups* das tarefas pertencentes a p_j (e que ainda não executaram), serão alocadas em novos processadores ou cores escolhidos pelo escalonamento tolerante a falhas. Neste caso, aplica-se dentro do código de cada tarefa *backup* a mesma decomposição OpenMP da sua tarefa primária como explicado no código híbrido. Para o caso de tarefas *backups* T_{kj} do mesmo nível k pode ser aplicada a agregação OpenMP em um mesmo processador p analisada antes. Espera-se que o tempo de execução de cada tarefa *backup* (T_{kk} ou T_{kj}) seja menor que o da mesma *backup* executando com a abordagem MPI anterior [Sardina 2010] ou custos de comunicação sejam reduzidos dependendo do caso.

5. Ambiente de Testes

Inicialmente a proposta foi testada em uma única máquina Intel Core i7 e sistema operacional Linux, com o objetivo de observar o funcionamento da implementação com a nova abordagem híbrida. Depois deve ser estendida a outras arquiteturas maiores. O objetivo é pesquisar em distintas arquiteturas com características multicores e de maior escala, para assim analisar hierarquias de memórias com seus efeitos ou benefícios para a execução das aplicações na presença de falhas. Por exemplo, realizar estudos em relação às hierarquias de memória, vertical e horizontal, a avaliação do desempenho da comunicação MPI pela troca de mensagens nos *clusters*, ao gerenciamento de caches paralelas hierárquicas, compartilhamento de caches de diferentes níveis e suas limitações entre outros aspectos. Desta forma, propor novos algoritmos multicores que maximizem a localidade dos dados e explorem a estrutura hierárquica das caches.

Os estudos comparativos no ambiente tem como objetivo analisar quanto muda o desempenho da aplicação com a nova abordagem híbrida, sem falha e/ou com falha. Para testar a tolerância a falhas inicialmente é simulada a ocorrência de uma falha permanente de processador mediante a execução de um arquivo *script* que contêm a chamada ao comando *killall*, para matar os processos que formam parte do processador com falha. Este arquivo pode ser disparado enquanto a aplicação ainda está sendo executada. Os testes estão sendo realizados com o ambiente de execução em modo exclusivo e principalmente

para 2 cenários de execução da aplicação: com a ferramenta sem falha e com a ferramenta com falha para as duas abordagens comparadas. Medições de tempo de execução e do número de mensagens são registradas, variando o número de tarefas.

Para avaliar a proposta os próximos testes serão realizados em três ambientes maiores diferentes e com outras aplicações alvo modeladas como descrito na seção 2.1. O primeiro ambiente é uma rede ethernet Linux de computadores no laboratório da ECT, o segundo uma rede WiFi de *notebooks* e o último o *Cluster* SGI Altix do Instituto Internacional de Física da UFRN.

6. Discussões e Trabalhos Futuros

A programação híbrida com o MPI e o OpenMP, adequa-se às arquiteturas atuais baseadas em *clusters* multiprocessores. Permite diminuir o número de comunicações entre os diferentes nós e aumentar o desempenho de cada nó sem que ocasione um incremento considerável dos requisitos de memória. Aplicações que possuem dois níveis de paralelismo podem utilizar processos MPI para explorar paralelismo de granularidade grossa/média, trocando mensagens ocasionalmente para sincronizar informação e/ou distribuir trabalho, e utilizar *threads* para explorar paralelismo de granularidade média/fina por partilha do espaço de endereçamento. Aplicações que tenham restrições ou requisitos que possam limitar o número de processos MPI que podem ser usados podem tirar partido do OpenMP para explorar o poder computacional dos restantes processadores disponíveis. Aplicações cujo balanceamento de carga seja difícil de conseguir apenas por utilização de processos MPI, podem tirar partido do OpenMP para equilibrar esse balanceamento, atribuindo um diferente número de *threads* a diferentes processos MPI em função da respectiva carga.

Em resultados preliminares obtidos para a aplicação de Gauss, observa-se que o tempo de execução e o número de mensagens diminuim com o incremento do número de tarefas. A execucão da aplicação com a nova abordagem obteve melhor desempenho quando comparada com resultados com a abordagem MPI anterior não híbrida. A aplicação com a ferramenta modificada, com falha e sem falha obteve maior desempenho quando comparado com o cenário da abordagem anterior. Portanto, nesta análise inicial, métricas como tempo de execução e número de mensagens apresentam uma diminuição nos experimentos realizados variando a quantidade de tarefas.

Nesta etapa inicial estuda-se quanto a estratégia é viável sobre uma arquitetura multicore menor e como diversos fatores podem influenciar no desempenho da aplicação, tais como: o número de processos criados ao mesmo tempo, o número de tarefas escalonadas por processador e o número máximo de mensagens que a estrutura de memória usada pelo processador pode armazenar. A ferramenta híbrida implementada nesta primeira etapa tem um escopo menor. Uma segunda etapa será estendê-la a *clusters* e redes maiores, adaptada ao *middleware* de [Boeres and Rebello 2004, de P. Nascimento et al. 2005] e com variações do algoritmo de escalonamento de tolerância a falhas propostos. Em relação ao modelo de falhas, levar-se em consideração a ocorrência de múltiplas falhas.

Referências

(May, 2008). OpenMP application program interface version 3.0 complete specifications. http://www.openmp.org/mp-documents/specs30.pdf.

(Nov, 2009). MPI: A message-passing interface standard version 2.1. `www.mpi-forum.org/docs/mpi21-report.pdf`.

Al-Omari, R., Somani, A. K., and Manimaran, G. (2005). An adaptive scheme for fault-tolerant scheduling of soft real-time tasks in multiprocessor systems. *J. Parallel Distrib. Comput.*, 65(5):595–608.

Anne Benoit, M. H. and Robert, Y. (April 14-18, 2008). Fault tolerant scheduling of precedence task graphs on heterogeneous platforms. In *Proceedings of the 22th ACM/IEEE International Parallel Distributed Processing Symposium IPDPS'08 - APDCM'08 IEEE Computer Society Press*, Miami, Florida, USA.

Aversa, R., Di Martino, B., Rak, M., Venticinque, S., and Villano, U. (2005). Performance prediction through simulation of a hybrid MPI/OpenMP application. *Parallel Comput.*

Benoit, A., Hakem, M., and Robert, Y. (2008). Realistic models and efficient algorithms for fault tolerant scheduling on heterogeneous platforms. In *Parallel Processing, 2008. ICPP '08. 37th International Conference on*, pages 8–12, Portland, Oregon, USA.

Boeres, C. and Rebello, V. E. F. (2004). Easygrid: towards a framework for the automatic grid enabling of legacy MPI applications: Research articles. *Concurrency And Computation : Practice And Experience*, 16(5):425–432.

Chorley, M. J., Walker, D. W., and Guest, M. F. (2009). Hybrid message-passing and shared-memory programming in a molecular dynamics application on multicore clusters. In *Int. J. High Perform. Comput.*, pages 196–211.

da Silva, J. A. (2010). *Tolerância a Falhas para Aplicações Autônomas em Grades Computacionais*. PhD thesis, Instituto de Computação, Universidade Federal Fluminense, Niterói, RJ, Brasil.

de P. Nascimento, A., da C. Sena, A., da Silva, J. A., de C. Vianna, D. Q., Boeres, C., and Rebello, V. E. F. (2005). Managing the execution of large scale MPI applications on computational grids. *17th. International Symposium on Computer Architecture and High Performance Computing*.

Qin, X. and Jiang, H. (2006). A novel fault-tolerant scheduling algorithm for precedence constrained tasks in real-time heterogeneous systems. *Parallel Computing*, 32(5):331–356.

Rabenseifner, R., Hager, G., and Jost, G. (2009a). Communication characteristics and hybrid MPI/OpenMP parallel programming on clusters of multi-core smp nodes. *Proceedings of the Cray Users Group Conference 2009 (CUG 2009)*.

Rabenseifner, R., Hager, G., and Jost, G. (2009b). Hybrid MPI/OpenMP parallel programming on clusters of multi-core smp nodes.

Sardina, I. M. (2010). *Escalonamento Estático de Tarefas Bi-objetivo e Tolerante a Falhas para Sistemas Distribuídos*. PhD thesis, Instituto de Computação, Universidade Federal Fluminense, Niterói, RJ, Brasil.

Sardina, I. M., Boeres, C., and Drummond, L. M. A. (2011a). An efficient weighted bi-objective scheduling algorithm for heterogeneous systems. *Parallel Computing*, 37:349–364.

Sardina, I. M., Boeres, C., and Drummond, L. M. A. (2011b). Escalonamento estático bi-objetivo e tolerante a falhas para sistemas distribuídos. In *Simposio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, Campo Grande.

Su, M. F., El-Kady, I., Bader, D. A., and Lin, S. (2004). A novel fdtd application featuring openmp-mpi hybrid parallelization. In *Proceedings of the 2004 international Conference on Parallel Processing ICPP*, pages 373–379, Washington, DC, USA. IEEE Computer Society.

Topcuouglu, H., Hariri, S., and Wu, M. (2002). Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions Parallel Distributed Systems*, 13(3):260–274.

DifATo - Difusão Atômica Tolerante a Faltas Bizantinas Baseada em Tecnologia de Virtualização

Marcelo Ribeiro Xavier Silva[1], Lau Cheuk Lung[1], Aldelir Fernando Luiz[2,3], Leandro Quibem Magnabosco[1]

[1]Departamento de Informática e Estatística - Universidade Federal de Santa Catarina - Brasil

[2]Câmpus Avançado de Blumenau - Instituto Federal Catarinense - Brasil

[3]Departamento de Automação e Sistemas - Universidade Federal de Santa Catarina - Brasil

marcelo.r.x.s@posgrad.ufsc.br, lau.lung@inf.ufsc.br, aldelir@das.ufsc.br

leandro.magnabosco@posgrad.ufsc.br

Abstract. *This paper presents a byzantine fault-tolerant protocol for atomic multicast whose algorithm manages to implement a reliable consensus service with only $2f + 1$ servers to tolerate f faulty ones. For the creation of the algorithm we used common technologies such as virtualization and data sharing abstractions. The system model adopted is hybrid, meaning that the assumptions of synchrony and occurrence of faults consider each component separately. Moreover, in our model, we use two networks to provide the service, a payload network, where messages are exchanged between clients and servers, and a tamperproof network, where the messages are ordered.*

Resumo. *Este trabalho apresenta um protocolo de difusão atômica tolerante a faltas Bizantinas (Byzantine Fault Tolerant - BFT) em que o algoritmo implementa um serviço confiável de consenso com $2f + 1$ servidores tolerando até f faltosos. Para a criação do algoritmo são utilizadas tecnologias comuns como virtualização e abstrações de compartilhamento de dados. O modelo de sistema adotado é híbrido, o que significa que as premissas de sincronismo e ocorrência de faltas consideram cada componente separadamente. Além disso, em nosso modelo, utilizamos duas redes para fornecer o serviço, uma rede de carga, onde são trocadas mensagens entre os clientes e servidores, e uma rede inviolável, onde são feitas as ordenações das mensagens.*

1. Introdução

A dificuldade em construir sistemas distribuídos pode ser drasticamente reduzida através do uso de primitivas de comunicação em grupo, tal como a difusão atômica (ou difusão com ordem total) [Défago et al. 2004]. A difusão atômica assegura que mensagens enviadas para um conjunto de processos serão entregues por estes na mesma ordem, sendo empregada nos mais diversos domínios de aplicação como: sincronização de relógios, CSCW (*Computer Supported Cooperative Work*), memórias distribuídas, replicação de base de dados [Rodrigues et al. 1993, Kemme et al. 2003, Bessani et al. 2006], e é a base para abordagens de replicação de máquina de estados [Schneider 1990], e também o componente principal de muitos sistemas tolerantes a faltas [Castro and Liskov 2002, Yin et al. 2003, Correia et al. 2006, Favarim et al. 2007].

A literatura nos mostra uma quantidade considerável de trabalhos sobre difusão com ordem total, com as mais variadas abordagens e algoritmos. Entretanto, em sua maioria, estes algoritmos consideram modelos de sistema sujeitos apenas a faltas de parada (i.e. *crash*) [Défago et al. 2004, Ekwall et al. 2004], de modo que poucos são os trabalhos que endereçam faltas arbitrárias/Bizantinas [Correia et al. 2006, Reiter 1994]. Em geral, as abordagens usam algoritmos de consenso para estabelecer um acordo acerca da ordenação das mensagens, e necessitam de, pelo menos, $3f + 1$ processos envolvidos no procedimento. Por outro lado, alguns trabalhos propõem a separação do consenso do acordo, o que dá origem a um serviço de consenso [Guerraoui and Schiper 2001, Pieri et al. 2010].

Neste trabalho apresentamos o DifATo (acrônimo para **Dif**usão **A**tômica **To**lerante a Faltas Bizantinas), um serviço de consenso com o propósito de difundir as mensagens atomicamente, a despeito de faltas Bizantinas. O modelo de sistema, bem como a arquitetura que propomos necessita apenas de $2f + 1$ servidores para compor o serviço de consenso e baseia-se em um modelo híbrido, isto é, um modelo onde variam, de componente para componente, as suposições sobre sincronismo e presença/severidade de faltas e falhas [Correia et al. 2002, Veríssimo 2006, Correia et al. 2004]. Em nosso modelo consideramos a existência de uma rede de carga (i.e. *payload*), que é utilizada para a comunicação entre os clientes e o serviço de consenso; e também uma rede inviolável, onde os servidores executam a ordenação das mensagens. Nossa contribuição vem ao encontro da proposição de melhorias no serviço de consenso, a fim de torná-lo tolerante a faltas Bizantinas e com um custo mais reduzido (i.e. com apenas $2f + 1$ servidores).

O restante deste artigo está organizado da seguinte forma. A Seção 2 apresenta os trabalhos relacionados; a Seção 3 descreve o modelo de sistema adotado para este trabalho; na Seção 4 são apresentados os algoritmos que compõem a proposta, e uma breve descrição das provas para os algoritmos são apresentadas na Seção 5. Na Seção 6 se apresenta alguns aspectos de implementação e os resultados obtidos, e por fim, a Seção 7 conclui o artigo.

2. Trabalhos Relacionados

O problema de difusão atômica tem sido amplamente estudado nas últimas décadas [Rodrigues et al. 1993, Reiter 1994, Veríssimo 2006, Kemme et al. 2003, Ekwall et al. 2004, Correia et al. 2006]. Em sua maioria, as abordagens consideram que os processos podem sofrer apenas por faltas de parada, isto é, não considerando para tanto, faltas de natureza arbitrária ou Bizantina [Lamport et al. 1982].

Em [Reiter 1994] é apresentado o Rampart, um suporte para difusão atômica confiável em sistemas sujeitos a faltas Bizantinas. O algoritmo deste é baseado em um serviço de associação a grupo, requerendo que pelo menos um terço de todos os processos pertencentes a visão atual entrem em acordo sobre a exclusão de alguns processos do grupo. A difusão atômica é feita por um membro do grupo chamado de sequenciador, cuja responsabilidade é determinar a ordem para as mensagens enviadas na visão atual. Na próxima visão, outro sequenciador é escolhido por um algoritmo determinístico. O Rampart assume um modelo de sistema assíncrono, com canais FIFO confiáveis, e uma infraestrutura de chaves públicas conhecida por todos os processos. Com a suposição de canais de comunicação autenticados, a integridade das mensagens trocadas entre dois processos não Bizantinos é garantida.

Guerraoui e Schiper propuseram um serviço genérico de consenso [Guerraoui and Schiper 2001] para resolver problemas de acordo, dentre os quais

está incluída a difusão atômica. Este serviço constitui a base para nossa proposta. Neste caso, o modelo proposto pelos autores considera um ambiente sujeito a apenas falhas por parada, o qual possui um serviço de consenso que separa o consenso do problema de acordo a ser resolvido. O sistema requer que se tenha um detector de falhas perfeito [Chandra and Toueg 1996] (baseando o serviço de consenso) e a resiliência varia de acordo com o desempenho desejado.

Alguns anos mais tarde, Correia e Veríssimo mostraram uma transformação de consenso para difusão atômica [Correia et al. 2006], em que o modelo de sistema apresentado assumia um ambiente Bizantino, no qual $f = \lfloor \frac{(n-1)}{3} \rfloor$ faltas eram toleradas. Os autores implementaram um protocolo de consenso multivalorado sobre um consenso binário aleatório, e um protocolo de difusão confiável. O protocolo de difusão atômica é criado através de sucessivas transformações a partir do protocolo de consenso. A difusão atômica é feita através do uso de um vetor de *hashes*. Cada processo do sistema propõe um valor para o vetor de consenso (que é o vetor com os *hashes* das mensagens). O protocolo de vetor de consenso decide sobre um vetor X_i com pelo menos $2f + 1$ vetores H de diferentes processos. Em seguida, as mensagens são armazenadas em um conjunto para serem atomicamente entregues na ordem pré-estabelecida.

Mais recentemente, Pieri et al. propôs uma extensão ao serviço genérico de consenso para ambientes Bizantinos [Pieri et al. 2010]. O modelo de sistema proposto possuía $n_c = 3f_c + 1$ clientes e $n_s = 2f_s + 1$ servidores, e fazia uso de máquinas virtuais para prover o serviço genérico de consenso. Na proposta dos autores, o consenso atômico se inicia sempre que um dos processos, conhecido no protocolo por iniciador, difunde de maneira confiável uma mensagem m_i para o conjunto de clientes. Ao receber a mensagem m_i, cada cliente envia uma proposta de ordenação de m_i para o serviço genérico de consenso. Quando os servidores recebem $n_c - f_c$ propostas de clientes para a mesma instância de consenso, cada servidor inicia o protocolo de consenso, e então o resultado do protocolo é enviado aos clientes. A importância deste trabalho para a literatura, é que ele foi o primeiro a tornar o serviço genérico de consenso disponível para ambientes sujeitos a faltas Bizantinas, a despeito do número de clientes ser limitado. Além disso, o sistema precisava lidar com o a existência de clientes faltosos, diminuindo assim, a resiliência do sistema. De um ponto de vista, isto é aceitável quando se pretende trabalhar com problemas genéricos de acordo, porém, em se tratando especificamente do problema de difusão atômica, é algo que se torna custoso.

3. Modelo de Sistema e Arquitetura

O modelo de sistema adotado é híbrido [Veríssimo 2006], o que significa que existe variação, de componente para componente, em relação às suposições de sincronismo e presença/severidade de faltas e falhas [Correia et al. 2002, Veríssimo 2006]. Em nosso modelo, consideramos diferentes suposições para os subsistemas que executam no *host* e no *guest* das máquinas virtuais que compõem o sistema. Neste modelo, o conjunto $C = \{c_1, c_2, c_3, ...\}$ representa o número finito de processos clientes e $S = \{s_1, s_2, s_3, ..., s_n\}$ representa o conjunto de servidores contendo n elementos que implementam o serviço de consenso. Cada servidor possui uma máquina virtual que contém apenas um sistema como *guest*. O modelo de falhas admite que um número finito de clientes pode sofrer faltas por parada, e até $f \leq [\frac{n-1}{2}]$ servidores podem falhar em suas especificações apresentando comportamento arbitrário ou Bizantino [Lamport et al. 1982]: um processo faltoso pode desviar de suas especificações omitindo ou parando de enviar mensagens, ou ainda apresentar

qualquer tipo de comportamento (malicioso ou não) não especificado. Todavia, assumimos a independência de falhas, de tal maneira que a ocorrência de uma falta em um determinado servidor, é independente da ocorrência da mesma falta em outro servidor. Na prática, isto é possível por meio do uso extensivo de diversidade (diferentes *hardware/software*, sistemas operacionais, máquinas virtuais, bases de dados, linguagens de programação, etc) [Obelheiro et al. 2005].

Nosso modelo de sistema prevê o uso de duas redes de comunicação. A primeira, que é a rede de carga (ou *payload*) é assíncrona e é utilizada para transferência de dados da aplicação, isto é, para a interação entre os clientes e servidores. Assim, não fazemos quaisquer suposições baseadas em tempo para a rede de carga, de modo que sua utilização ocorre apenas para envio de requisições e respostas entre clientes e servidores. Por outro lado, a segunda rede, que implementa um serviço de Registradores Compartilhados Distribuídos (i.e. uma memória compartilhada) é controlada, e é utilizada apenas para a interação entre os servidores, para que estes possam trocar as mensagens do protocolo de consenso. Deste modo, para esta rede assumimos as seguintes hipóteses:

- possui um número finito e conhecido de membros;
- é segura e resistente a qualquer possível ataque, e pode falhar apenas por parada (*crash*);
- é capaz de executar operações com delimitação temporal;
- provê apenas duas operações, uma para leitura e outra para escrita, em registradores; estas operações não podem ser afetadas por faltas maliciosas.

Cada máquina física possui seu próprio espaço dentro dos registradores compartilhados distribuídos, e é neste espaço que cada máquina virtual registra as mensagens do tipo PROPOSE, ACCEPT e CHANGE. Todos os servidores podem escrever em todo o espaço de registradores, independente de quem tem o direito de escrita no mesmo.

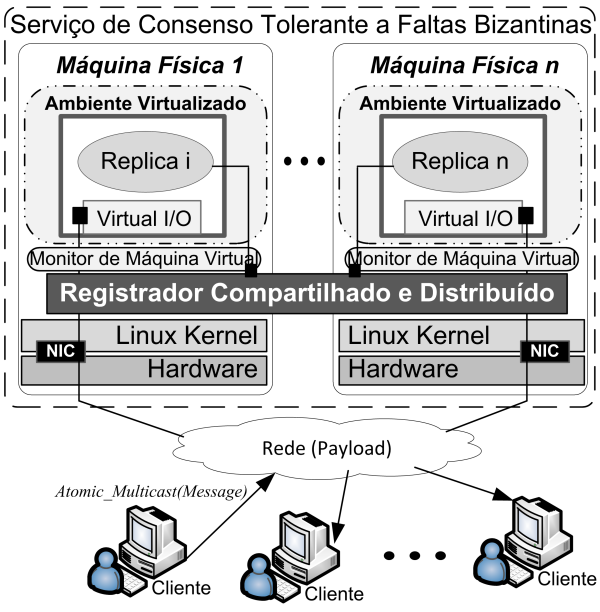


Figura 1. Visão geral da arquitetura.

Assumimos que cada par cliente-servidor c_i, s_j e cada par de servidores s_i, s_j está conectado por um canal confiável com duas propriedades: se o remetente e o destinatário de uma mensagem são ambos corretos, então (1) a mensagem será recebida em

algum momento e (2) a mensagem não é modificada no canal [Correia et al. 2004]. Na prática, estas propriedades podem ser obtidas com o uso de criptografia e retransmissão [Wangham et al. 2001]. Para isto, empregamos o uso de códigos de autenticação de mensagens (MACs) como *checksums* criptográficos, o que é viabilizado apenas pelo uso de criptografia simétrica [Menezes et al. 1996, Castro and Liskov 2002]. Não obstante, é requerido o compartilhamento de chaves simétricas pelos processos, a fim de permitir o uso de MACs, onde assumimos que estas chaves são distribuídas antes do protocolo ser executado. Em termos de implementação, isto pode ser resolvido usando protocolos de distribuição de chaves disponíveis na literatura [Menezes et al. 1996]. Todavia, salientamos que este problema está fora do escopo deste trabalho.

Assumimos que apenas as máquinas físicas podem, de fato, conectar-se a rede controlada usada pelos registradores. Isto significa que os registradores são acessíveis apenas pelas máquinas físicas que compõe o sistema e hospedam máquinas virtuais. Com isso, o acesso aos registradores não é possível através do acesso às máquinas virtuais. Cada processo é encapsulado em sua própria máquina virtual, de modo a assegurar o isolamento. Toda a comunicação cliente-servidor acontece em uma rede separada (rede de carga) e, do ponto de vista dos clientes, a máquina virtual é transparente. Portanto, os clientes não são capazes de identificar a arquitetura físico-virtual. Cada máquina possui apenas uma placa de interface de rede (NIC), o hospedeiro utiliza *firewall* e/ou modo *bridge* para assegurar a divisão das redes. Assumimos que as vulnerabilidades do hospedeiro não podem ser exploradas através da máquina virtual. O monitor da máquina virtual assegura o isolamento, garantindo que um atacante não tem meios para acessar o hospedeiro através da máquina virtual. Esta é uma característica presente em grande parte das tecnologias de virtualização mais comuns, tal como VirtualBox, LVM, XEN, VMWare, VirtualPC, etc. Nosso modelo assume que o sistema hospedeiro é inacessível externamente, o que é também garantido pelo uso do modo *bridge* e/ou *firewall* no sistema hospedeiro.

3.1. Registradores Compartilhados Distribuídos (RCD)

A memória compartilhada emulada consiste em uma abstração de registradores disponível para um conjunto de processos, na qual a comunicação subjacente é realizada através de troca de mensagens [Guerraoui and Rodrigues 2006]. Esta definição é realmente atraente, pois permite que a memória compartilhada seja construída utilizando qualquer tecnologia para compartilhamento de memória. A memória compartilhada, emulada ou não, pode ser vista como um *array* de registradores compartilhados, em que consideramos a definição sob o ponto de vista do programador. O tipo do registrador compartilhado especifica quais operações podem ser realizadas e os valores retornados pela operação [Guerraoui and Rodrigues 2006]. Os tipos mais comuns são os de leitura/escrita. As operações dos registradores são invocadas pelos processos do sistema para troca de informações. Para a realização deste trabalho, criamos uma abstração de memória emulada compartilhada, que denominamos como Registradores Compartilhados Distribuídos (RCD). Esta abstração é baseada em troca de mensagens através de uma rede controlada, que faz uso de arquivos locais para efetuar as operações. Assumimos que a rede controlada é acessível apenas por componentes do RCD. Os registradores são implementados nos hospedeiros das máquinas virtuais, onde assumimos que o monitor da máquina virtual assegura o isolamento entre eles e seus convidados.

O acesso aos RCDs é realizado por meio de apenas duas operações:

1. $read()$ - Usada para ler a última mensagem escrita nos RCDs;

2. $write(m)$ - Usada para escrever a mensagem m nos RCDs.

A execução destas operações leva em consideração duas propriedades básicas, que são:

(i) Vivacidade (*liveness*) - A operação eventualmente termina;
(ii) Segurança (*safety*) - A operação de leitura sempre retorna o último valor escrito.

Em termos de implementação, para tornar possível o atendimento destas propriedades, em cada servidor é criado um arquivo onde o convidado tem acesso apenas para escrita e outro onde o convidado tem acesso apenas para leitura, e todos os acessos são feitos por um único processo [Guerraoui and Rodrigues 2006]. Os RCDs aceitam apenas mensagens que estejam de acordo com a especificação do protocolo, isto é, tipificadas em que se admite apenas os tipos: (*i*) PROPOSE, (*ii*) ACCEPT e (*iii*) CHANGE. Com isso, todas as mensagens que não seguem esta especificação são ignoradas e descartadas.

De outro modo, assumimos que a comunicação nos RCDs se dá através de canais do tipo *fair links*, os quais atendem as condições de que, se ambos o remetente e o destinatário de uma mensagem são corretos, então [Yin et al. 2003]:

1. Se uma mensagem for enviada infinitas vezes para um destinatário, então a mensagem é recebida infinitas vezes;
2. Existe um atraso T de modo que, se uma mensagem é retransmitida infinitas vezes para um destinatário a partir de um tempo t_0, então o destinatário receberá a mensagem pelo menos uma vez antes de $t_0 + T$;
3. As mensagens não são modificadas no canal.

Entendemos que estas suposições são bastante razoáveis na prática, já que os RCDs são implementados em uma rede síncrona e separada, e que sofre apenas faltas por parada - baseado no isolamento provido pelo monitor de máquinas virtuais.

3.2. Propriedades da Difusão Atômica

O problema de difusão atômica, ou difusão confiável com ordem total, consiste em garantir a entrega de um conjunto de mensagens, na mesma ordem, para todos os processos que fazem parte de um sistema. A definição em um contexto Bizantino pode ser feita, considerando as seguintes propriedades:

DA1 *Validade* - Se um processo correto difunde uma mensagem m, então algum processo correto eventualmente entrega m.
DA2 *Acordo* - Se um processo correto entrega uma mensagem m, então todos os processos corretos eventualmente entregam m.
DA3 *Integridade* - Para qualquer mensagem m, todo processo correto entrega m no máximo uma vez, e se o remetente de m for correto, então m foi anteriormente difundida por este remetente.
DA4 *Ordem total* - Se dois processos corretos entregam duas mensagens com os prefixos m_{i-1} e m_i, então ambos os processos entregam as duas mensagens de maneira que m_{i-1} antecede m_i.

4. Algoritmo DifATo

Discussão: O procedimento necessita que apenas um servidor atue como sequenciador. Este servidor é responsável por propor as ordens para as mensagens dos clientes. Os demais servidores são apenas réplicas do serviço. Inicialmente o sequenciador é o processo com o menor número identificador (zero). A mudança de sequenciador acontece sempre que a maioria dos servidores ($f+1$) concordarem que esta condição é necessária. Como em outros sistemas tolerantes a faltas Bizantinas [Correia et al. 2004, Castro and Liskov 2002, Yin et al. 2003], é necessário lidar com o problema de um servidor p_j malicioso que pode descartar mensagens dos clientes. Em função disto, os clientes enviam suas mensagens para serem ordenadas, e esperam recebê-las de volta devidamente ordenadas, em até um tempo $T_{reenviar}$. Depois de passado este tempo, o cliente envia sua mensagem para todos os servidores. Um servidor correto, quando recebe uma mensagem do cliente e não é um sequenciador, solicita uma mudança de sequenciador. Se $f + 1$ servidores solicitam uma mudança de sequenciador, então os servidores corretos efetuam a mudança e o protocolo prossegue. Entretanto, a rede de carga é assumidamente assíncrona, por isso, não existem limites para os atrasos na comunicação, e não é possível definir um valor ideal para $T_{reenviar}$. Correia [Correia et al. 2004] mostra que o valor de $T_{reenviar}$ envolve uma troca: se o valor for muito alto, o cliente pode esperar demais pela ordenação da mensagem; se baixo demais, o cliente pode reenviar a mensagem sem necessidade. O valor deve considerar essa troca. Se a mensagem é reenviada sem necessidade, sua duplicata é descartada pelo sistema.

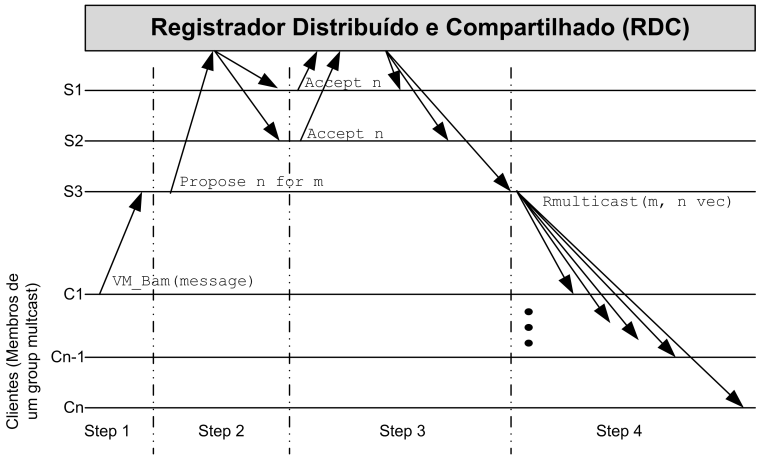


Figura 2. Fluxo da difusão atômica.

Esta seção oferece uma descrição mais aprofundada do algoritmo. A sequência de operações dos algoritmos é apresentada tanto no nó que atua como sequenciador, como naqueles que não desempenham este papel, isto é, os demais nós servidores. Primeiramente é considerada a operação do algoritmo com a ausência de faltas (i.e. caso normal) e, na sequencia, com a presença de faltas. O diagrama de fluxos da operação na ausência de faltas pode ser visto na Figura 2. Por clareza de apresentação, consideramos apenas um único grupo de difusão.

4.1. Operação em Caso Normal

Passo 1) O procedimento inicia-se quando algum cliente c_i envia a mensagem $\langle ORDER, m, t, v\rangle_{\sigma ci}$ para o sequenciador com sua mensagem m incluída. O

campo t é a marca de tempo da mensagem para assegurar a semântica de apenas uma ordenação por mensagem. Desta maneira os servidores só executam a ordenação de mensagens cuja marca de tempo seja maior que a anterior, para um mesmo cliente. O campo v é o vetor que gera um MAC por servidor, cada um obtido através da chave compartilhada entre clientes e servidores. Portanto, cada servidor pode testar a integridade da mensagem utilizando este vetor. Caso uma mensagem já tenha sido ordenada, o servidor apenas a reenvia para o cliente.

Algoritmo 1: Algoritmo executado pelo nó sequenciador.

```
   Constants:
   f : int // Maximum tolerated faults
   T : int // Maximum waiting time for a proposal to be decided
   Variables:
   accepted : int // counter of acceptance for some ordering
 1 upon receive ⟨ORDER, m, t_j, v⟩_σci from client
 2 if t_j ≤ t_{j-1} for c_i then
 3 |   if has(m) into buffer then
 4 |   |   rmulticast( getOrdered( m ) from buffer );
 5 |   end
 6 |   return;
 7 end
 8 if isWrong( v ) then
 9 |   return;
10 end
11 write( ⟨PROPOSE, n_o, ⟨ORDER, m, t_j, v⟩_σci, mac⟩_σsi ) into RCD;
12 accepted = waitForAcceptance( T );
13 if accepted ≥ f + 1 then
14 |   store ⟨PROPOSE, n_o, ⟨ORDER, m, t_j, v⟩_σci, mac⟩_σsi in the atomic buffer;
15 end
16 return;
```

Passo 2) Depois de verificar se o MAC em v é correto e se a marca de tempo é válida para a mensagem do cliente, o sequenciador gera uma mensagem $\langle PROPOSE, n_o, o, mac\rangle_{\sigma si}$ onde o representa a mensagem original do cliente, n_o é o número de ordenação para o e mac é o MAC gerado pelo sequenciador. Os RCDs automaticamente identificam a mensagem proposta com o id do sequenciador. O sequenciador espera pela aceitação dos demais servidores, isto é, f processos concordando com a proposta. Ao ter a mensagem aceita, o sequenciador salva a mensagem e a ordenação em seu *buffer*. Este comportamento pode ser observado no algoritmo 1. Como foi discutido, todas as mensagens escritas nos RCDs serão entregues se o destinatário e o remetente não sofreram uma parada (*crash*).

Passo 3) Ao receber uma proposta, o servidor s_k a valida: (i) s_k verifica, usando o vetor de MACs, se o conteúdo da mensagem m está correto e (ii) verifica se não existe outra proposta anteriormente aceita para o número de ordenação n. Depois de aceitar a proposta, s_k escreve uma mensagem $\langle ACCEPT, n_o, h_m, mac\rangle_{\sigma sk}$ nos registradores. Esta mensagem possui o *hash* da mensagem do cliente h_m, o número de ordenação aceito e o MAC mac gerado pelo servidor. Após escrever a mensagem de aceite, o processo aguarda por $f - 1$ mensagens de aceitação para, então, salvar a mensagem no *buffer*. Este comportamento pode ser visto no algoritmo 2.

Passo 4) O sequenciador difunde de maneira confiável a mensagem com o número de ordenação e um vetor de MACs assinado por pelo menos $f + 1$ servidores diferentes que aceitaram a ordem proposta. Após receber e validar o vetor, os clientes finalmente aceitam a mensagem e a entregam na ordem estipulada.

Algoritmo 2: Algoritmo executado pelo(s) nó(s) não sequenciador(es).

```
   Constants:
   f : int // Maximum tolerated faults
   T : int // Maximum waiting time for a proposal to be decided.
   Variables:
   accepted : int // counter of acceptance for some ordering
1  upon read ⟨PROPOSE, n_o, ⟨ORDER, m, t_j, v⟩_σci, mac⟩_σss from RCD
2  if isValid( v ) and isValid( n ) and t_j > t_j−1 for c_i then
3      write( ⟨ACCEPT, n_o, h_m⟩_σsi ) into RCD;
4      accepted = waitForAcceptance( T );
5      if accepted ≥ f + 1 then
6          store ⟨PROPOSE, n_o, ⟨ORDER, m, t_j, v⟩_σci, mac⟩_σss in the atomic buffer;
7      end
8  else
9      write( ⟨CHANGE, h_m, s_s⟩_σsi ) into RCD and bufferize;
10 end
11 return;
```

4.2. Operação em Situação com Faltas

A operação na presença de faltas implica que uma mudança de sequenciador ocorrerá, portanto, faremos uma breve explicação de como isso se desenvolve.

Mudança de sequenciador: Durante a configuração do sistema, todos os servidores recebem um número de identificação. Estes números são sequenciais e iniciam em zero. Todos os servidores conhecem o identificador S do sequenciador e o número total de servidores no sistema. Quando $f + 1$ servidores corretos suspeitam do sequenciador atual, eles simplesmente definem $S = S + 1$ como o próximo sequenciador, se $S < n - 1$, senão $S = 0$.

Ao validar uma proposta, o servidor s_k verifica, usando o MAC no vetor v, se o conteúdo da mensagem está correto. Se o conteúdo estiver correto e se o número de ordenação estiverem corretos, o servidor aceita a proposta. Caso contrário, s_k vai solicitar uma mudança de sequenciador:

1. Um servidor correto pode entrar em modo faltoso de operação de duas maneiras:
   (a) Quando o servidor s_k lê dos registradores uma mensagem de mudança de sequenciador, mas ainda não suspeita do servidor s_s. O servidor apenas armazena a mensagem em seu *buffer* local para utilização futura.
   (b) Se o processo s_k suspeita do sequenciador s_s com relação à mensagem m, então s_k escreve uma mensagem $\langle CHANGE, sid, h_m, s_s \rangle_{\sigma_{sk}}$ nos RCDs contendo o sid como seu próprio identificador, o *hash* da mensagem h_m que originou a suspeita e o identificador s_s do servidor em suspeita.
2. O servidor s_k inicia uma busca em seu *buffer* local, no intuito de encontrar $f + 1$ mensagens de mudança relacionadas à mensagem m e ao servidor s_s. Caso s_k encontre $f + 1$ (incluindo o próprio s_k) diferentes sid para a mesma mensagem, então o servidor efetua a mudança de sequenciador. Se o novo sequenciador for o próprio servidor s_k, então o servidor vai reiniciar a ordenação baseando-se nas mensagens já aceitas nos RCDs. Caso não seja s_k o novo sequenciador, s_k apenas aguarda pelas novas propostas.

Com a escolha de um novo sequenciador, o protocolo faz progresso como ocorre na operação normal, isto é, com a ausência de faltas.

5. Provas de Correção

Nesta seção demonstramos que os algoritmos apresentados neste trabalho satisfazem as propriedades especificadas na Seção 3.2. Mais precisamente, o protocolo de difusão

atômica é correto se satisfazer as propriedades definidas como **DA1** a **DA4**, vejamos:

Teorema 1 *O protocolo de difusão atômica especificado pelos Algoritmos 1 e 2 satisfaz as propriedades de* ***Validade*** *e* ***Acordo*** *(DA1 e DA2).*

Prova (esboço): Para esta prova, consideremos que um cliente correto envia uma mensagem ao sequenciador. Se o sequenciador é correto, então o mesmo irá escrever a mensagem no RCD, e a partir dai a mensagem estará disponível à todos os servidores, conforme se pode observar a partir do código das linhas 1 à 11, do Algoritmo 1. Após receber a proposta, cada servidor delibera se a mesma é válida, e quando a maioria concordar com a mesma, cada servidor correto vai armazená-la para, se necessário, difundi-la de maneira confiável, como pode ser visto nas linhas 12 à 15 do Algoritmo 1, e nas linhas 1 à 8 do Algoritmo 2. Caso o sequenciador não seja correto, o cliente irá reenviar sua mensagem para os demais servidores após o tempo $T_{reenviar}$. Com isso, os servidores corretos irão efetuar a mudança de sequenciador, e o novo sequenciador irá retomar a ordenação. E por fim, após receberem as mensagens ordenadas, os clientes corretos irão entregá-las na ordem estabelecida. □

Teorema 2 *O protocolo de difusão atômica especificado pelos Algoritmos 1 e 2 satisfaz a propriedade de* ***Integridade*** *(DA3).*

Prova (esboço): Pelo Algoritmo 1, especificamente nas linhas 2 à 7 é possível observar que o sequenciador efetua a ordenação das mensagens apenas uma vez, portanto, uma mensagem tem um, e somente um, valor de ordenação. Com isso, os clientes corretos entregam as mensagens apenas uma vez, e na ordem estipulada. E como pode ser visto nas linhas 8 à 10 do Algoritmo 1, e na linha 2 do Algoritmo 2, para garantir que uma mensagem só pode ter sido difundida por seu remetente, o vetor de MACs enviado com a mensagem é validado por cada servidor, a fim de evitar que um cliente ou um servidor possa se passar por outro cliente. Estas asserções provam a propriedade de integridade. □

Teorema 3 *O protocolo de difusão atômica especificado pelos Algoritmos 1 e 2 satisfaz a propriedade de* ***Ordem total*** *(DA4).*

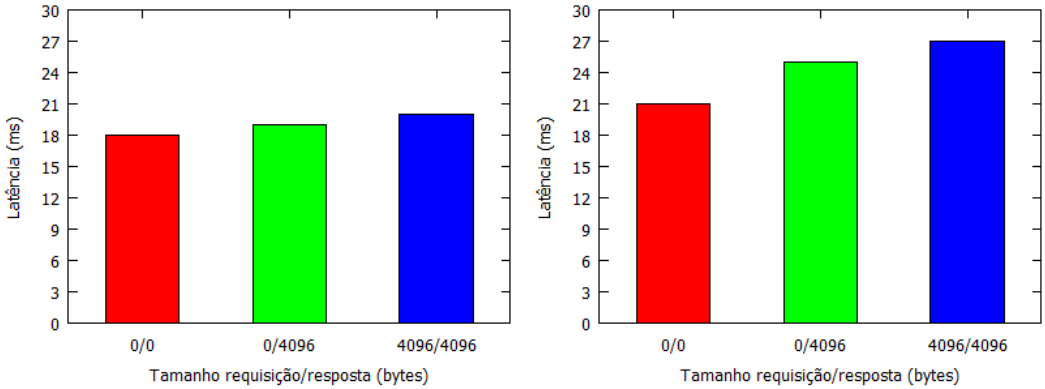
Prova (esboço): Pelo que se pode verificar nas linhas 11 à 15 do Algoritmo 1, e 1 à 8 do Algoritmo 2, respectivamente, se observa que o sequenciador apenas efetua a difusão confiável da mensagem, após ter sido executado o consenso. As mensagens difundidas são aquelas em que a ordenação foi aceita por pelo menos $f+1$ servidores, e a ordem de entrega é determinística. Como o sequenciador difunde a mensagem de maneira confiável, juntamente com sua ordenação e o vetor de MACs para atestar que aquela ordem é aceita pela maioria, consequentemente todos os processo corretos entregarão a mensagem na ordem estipulada, o que prova a propriedade em questão. □

6. Implementação, Avaliação e Resultados

Os algoritmos foram implementados usando a linguagem Java, com o uso do JDK 1.6.0. Os canais de comunicação foram implementados usando *sockets* TCP da API NIO. Os sistemas operacionais usados como hospedeiros das máquinas virtuais foram o "MacOSx Lion 10.7.4" e "Ubuntu 12.04", tendo como o monitor de máquinas virtuais o VirtualBox. Nos convidados (i.e. *guests*) das máquinas virtuais utilizou-se o "Ubuntu 12.04" e o "Debian 6

Stable". Para a avaliação do desempenho, escolhemos a métrica baseada na latência, dado que esta é a métrica largamente empregada na avaliação de sistemas computacionais, por representar de maneira simples, a eficiência do sistema [Jain 1991, Castro and Liskov 2002, Yin et al. 2003, Correia et al. 2006, Favarim et al. 2007].

Os valores foram obtidos através de *micro-benchmarks* com diferentes cargas. A latência foi obtida pela medida do tempo de ida e volta da comunicação (ou *round-trip*), o qual foi extraído pela medida do tempo entre o envio e o recebimento de um grupo de mensagens. O raciocínio por trás do uso de *micro-benchmarks* é medir adequadamente o algoritmo sem considerar influências externas. E a fim de avaliar a capacidade do protocolo, executamos as simulações com diferentes tamanhos de mensagens.



(a) Operação normal (b) Operação com faltas.

Figura 3. Desempenho verificado para o DifATo.

E tendo por finalidade a avaliação do desempenho do algoritmo na ausência de faltas, executou-se o protocolo em condições normais, e enviando 10.000 requisições através de um único cliente, e com três condições de carga: 0/0KB, 0/4KB e 4/4KB. Com isto temos: uma requisição vazia e uma resposta vazia, uma requisição vazia e uma resposta de 4KB de tamanho, e uma requisição de 4KB com uma resposta de 4KB. E para avaliar o algoritmo também em situações de falha, as réplicas foram configuradas para que, quando assumissem o papel de sequenciadores, enviassem uma entre dez propostas incorretas. As Figuras 3(a) e 3(b) apresentam a latência para cada experimento com as diferentes condições de carga. A latência foi obtida pelo cômputo da média entre o tempo observado após as respostas de todas as requisições enviadas. Como podemos observar, a latência apresenta variações mínimas entre diferentes cargas. Isso se explica pelo fato de que, para aumentar a eficiência do protocolo, o acordo é realizado com base em resumos criptográficos (*hash*) das mensagens, de modo que elas são difundidas apenas em dois passos de comunicação, isto é, no primeiro e no último (Figura 2).

Por fim, para avaliar de eficiência do protocolo proposto de maneira analítica, realizamos um estudo comparativo entre o DifATo e o estado da arte em sistemas de difusão atômica. Estes dados são apresentados na tabela 1. É importante salientar que todos os dados consideram apenas as execuções dos protocolos no caso normal, isto é, na ausência de faltas (mesmo no caso dos clientes). Caso se considere clientes faltosos a troca de mensagens em nosso protocolo aumenta para ${n_s}^2 + n_c$. Pelo dados podemos verificar que os

benefícios do uso do DifATo são visíveis quando comparados os números de passos de comunicação, quantidade de servidores necessário, e o número de mensagens trocadas. Nossa abordagem tem a melhor resiliência prática em termos de quantidade de servidores, além disso, são necessários menos passos para realizar a difusão atômica. Também, ao evitar o envolvimento dos clientes, permitimos que um número finito de clientes seja suscetível a falhas de parada.

Tabela 1. Comparação entre as propriedades de protocolos de difusão atômica.

Protocolos Avaliados	***Propriedades e características verificadas***			
	Resiliência	**Passos de comunicação**	**Mensagens trocadas**	**Tipo de faltas**
Rampart [Reiter 1994]	$3f+1$	6	$6n-6$	Bizantina
Guerraoui e Schiper [Guerraoui and Schiper 2001]	-	5	$3n_c+2n_s-3$	parada
Correia e Veríssimo [Correia et al. 2006]	$3f+1$	-	$18n^2+13n+1+16n^2f+10nf$	Bizantina
Pieri et al. [Pieri et al. 2010]	$3f_c+1+2f_s+1$	5	$2(ns^2+3nc-ns-1)$	Bizantina
DifATo	$2f+1$	4	${n_s}^2-n_s+n_c+1$	Bizantina

7. Conclusão

Ao explorar o uso dos registradores compartilhados distribuídos e da tecnologia de virtualização, foi possível propor uma rede inviolável para implementar um protocolo de suporte à difusão atômica tolerante a faltas Bizantinas. Neste sentido, foi mostrado que é possível implementar um serviço de consenso confiável com apenas $2f+1$ servidores a partir do uso de tecnologias comuns, tal como virtualização e abstração de compartilhamento de dados. A tecnologia de virtualização é amplamente utilizada e entrega o isolamento necessário entre os servidores e o exterior, da mesma forma que o uso dos RCDs torna bastante simples a manutenção das propriedades do protocolo. Como trabalhos futuros, está prevista a criação de um mecanismo semelhante ao que é apresentado em [Veronese et al. 2009], a fim de que seja possível tolerar também, faltas arbitrárias oriundas dos clientes.

Referências

Bessani, A. N., da Silva Fraga, J., and Lung, L. C. (2006). Bts: A byzantine fault-tolerant tuple space. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 429–433. ACM.

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

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

Correia, M., Neves, N. F., and Verissimo, P. (2004). How to tolerate half less one byzantine nodes in practical distributed systems. In *Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on*, pages 174–183. IEEE.

Correia, M., Neves, N. F., and Veríssimo, P. (2006). From consensus to atomic broadcast: Time-free byzantine-resistant protocols without signatures. *The Computer Journal*, 49(1):82–96.

Correia, M., Verissimo, P., and Neves, N. (2002). The design of a cots real-time distributed security kernel. *Dependable Computing EDCC-4*, pages 634–638.

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

Ekwall, R., Schiper, A., and Urbán, P. (2004). Token-based atomic broadcast using unreliable failure detectors. In *Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on*, pages 52–65. IEEE.

Favarim, F., Fraga, J. S., Lung, L. C., Correia, M., and Santos, J. F. (2007). Exploiting tuple spaces to provide fault-tolerant scheduling on computational grids. In *Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC'07. 10th IEEE International Symposium on*, pages 403–411. IEEE.

Guerraoui, R. and Rodrigues, L. (2006). *Introduction to reliable distributed programming*. Springer-Verlag New York Inc.

Guerraoui, R. and Schiper, A. (2001). The generic consensus service. *Software Engineering, IEEE Transactions on*, 27(1):29–41.

Jain, R. (1991). *The art of computer systems performance analysis*, volume 182. John Wiley & Sons Chichester.

Kemme, B., Pedone, F., Alonso, G., Schiper, A., and Wiesmann, M. (2003). Using optimistic atomic broadcast in transaction processing systems. *Knowledge and Data Engineering, IEEE Transactions on*, 15(4):1018–1032.

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

Menezes, A. J., Van Oorschot, P. C., and Vanstone, S. A. (1996). *Handbook of applied cryptography*. CRC.

Obelheiro, R., Bessani, A., and Lung, L. (2005). Analisando a viabilidade da implementação prática de sistemas tolerantes a intrusões. In *Anais do V Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais - SBSeg'05*, pages 99–112. SBC.

Pieri, G., da Silva Fraga, J., and Lung, L. C. (2010). Consensus service to solve agreement problems. In *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, pages 267–274. IEEE.

Reiter, M. K. (1994). Secure agreement protocols: Reliable and atomic group multicast in rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 68–80. ACM.

Rodrigues, L., Veríssimo, P., and Casimiro, A. (1993). Using atomic broadcast to implement a posteriori agreement for clock synchronization. In *Proceedings of the 12th Symposium on Reliable Distributed Systems - SRDS'93*, pages 115–124. IEEE.

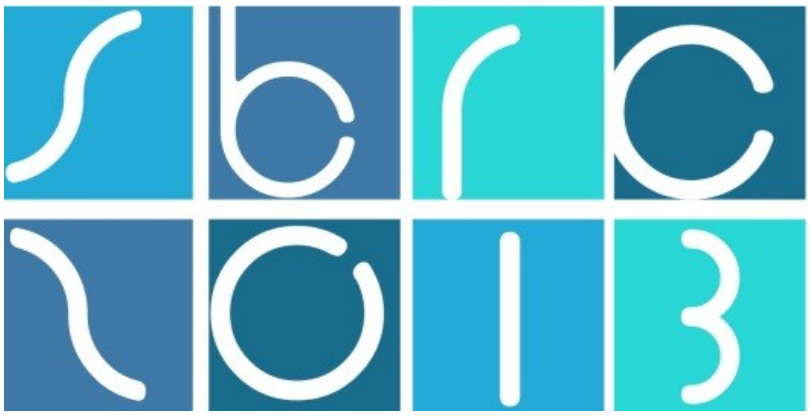
Schneider, F. B. (1990). Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319.

Veríssimo, P. E. (2006). Travelling through wormholes: a new look at distributed systems models. *ACM SIGACT News*, 37(1):66–81.

Veronese, G. S., Correia, M., Bessani, A. N., and Lung, L. C. (2009). Spin one's wheels? byzantine fault tolerance with a spinning primary. In *Reliable Distributed Systems, 2009. SRDS'09. 28th IEEE International Symposium on*, pages 135–144. IEEE.

Wangham, M. S., Lung, L. C., Westphall, C. M., and da Silva Fraga, J. (2001). Integrating ssl to the jacoweb security framework: Project and implementation. In *Integrated Network Management'01*, pages 779–792.

Yin, J., Martin, J., Venkataramani, A., Alvisi, L., and Dahlin, M. (2003). Separating agreement from execution for byzantine fault tolerant services. *ACM SIGOPS Operating Systems Review*, 37(5):253–267.



31º Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos
Brasília-DF

XIV Workshop de Testes e Tolerância a Falhas

Palestra Convidada

MapReduce Tolerante a Falhas Bizantinas

Luciana Arantes[1]

[1] Laboratoire d'Informatique de Paris 6 (LIP6)
Universidade Pierre et Marie Curie (Paris 6)

Biografia da Palestrante. *Luciana Arantes é professora (Maítre de Conférences) na Universidade Pierre et Marie Curie (Paris 6) desde 2001 e membro do projeto Regal, uma cooperação entre os laboratórios LIP6 e INRIA. Ela é graduada em Ciência da Computação pela Unicamp, Mestre em Engenharia Elétrica pela Escola Politécnica da Universidade de São Paulo e Doutora em Ciência da Computação pela Universidade de Paris 6 (2000). Sua área de pesquisa se concentra em algoritmos e sistemas distribuídos hierárquicos ou tolerantes a falhas para arquiteturas em grande escala, redes sem Fio e móveis, Grids e Clouds.*

Resumo. *MapReduce consiste em um modelo de programação e sistema de execução (framework) distribuído proposto pelo Google para o processamento de grandes volume de dados. Inspirado pelas funções map e reduce, usadas comumente em programação funcional, o modelo permite dividir o trabalho em um conjunto de tarefas independentes, executadas de forma paralela e distribuída pelo framework. Este também é responsável por prover tolerância a falhas, escalonamento de tarefas, distribuição de dados e balanceamento de carga. Entretanto, o mecanismo de tolerância a falhas oferecido pelo MapReduce suporta apenas falhas por parada ("crashes") de máquinas e tarefas ou corrupção de dados de arquivos, não tolerando falhas arbitrarias (bizantinas). Explorando o mecanismo de duplicação de tarefas e comparação dos resultados, esta palestra apresenta um sistema de execução MapReduce tolerante a falhas bizantinas de tarefas. Este foi desenvolvido no contexto do projeto FT-GRID (Programa PESSOA, Egide), uma cooperação entre a Universidade de Lisboa (Laboratório LASIGE) e a Universidade de Paris 6 (Laboratório LIP6).*

Índice por Autor