Wagner Saback Dantas

Um Arcabouço de Avaliação de Algoritmos de Sistemas de Quóruns Bizantinos

FLORIANÓPOLIS
2006

UNIVERSIDADE FEDERAL DE SANTA CATARINA
CURSO DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Um Arcabouço de Avaliação de Algoritmos de Sistemas de Quóruns Bizantinos

Dissertação submetida à
Universidade Federal de Santa Catarina
como parte dos requisitos para a
obtenção do grau de Mestre em Engenharia Elétrica.

Wagner Saback Dantas

Florianópolis, Agosto de 2006.

Um Arcabouço de Avaliação de Algoritmos de Sistemas de Quóruns Bizantinos

Wagner Saback Dantas

‘Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Engenharia Elétrica, Área de Concentração em *Controle, Automação e Informática Industrial*, e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Santa Catarina.’

_____

Prof. Joni da Silva Fraga, Dr.
Orientador

_____

Prof. Nelson Sadowski, Dr.
Coordenador do Programa de Pós-Graduação em Engenharia Elétrica

Banca Examinadora:

_____

Joni da Silva Fraga, Dr. – UFSC
Presidente

_____

Jean-Marie Farines, Dr. – UFSC

_____

Lau Cheuk Lung, Dr. – PUC-PR

_____

Magnos Martinello, Dr. – UFES

_____

Mario Dantas, Dr. – UFSC

A Ele que nos observa calmamente em sua onipresente e infinita bondade. A todos que, de alguma maneira, construíram o presente trabalho.

Agradecimentos

Fica difícil determinar neste espaço o grande contigente de pessoas que auxiliaram com um largo matiz de gestos na construção desta dissertação. Em um esforço de memória, em retrospecto narrativo, confiarei ao risco de esquecer-me de uma amostragem (já digo de antemão) de sujeitos. Todos os citados e não-citados doravante montaram o cotidiano meu em Desterro (Florianópolis) e, por tabela, influenciaram nas minhas vontade e representação que, por conseqüência, compuseram a concepção, forma e conteúdo deste trabalho. Do manifesto a seguir à conclusão. Da conclusão em diante.

Saúdo os brasileiros que, via CAPES (Ministério da Educação), agraciou-me com uma bolsa de estudos. Fernando e Graça, meus pais; Fábio, meu irmão; Thaís, minha sobrinha. Meus familiares queridos, em Salvador (Bahia), e em tantos outros locais. Amigos soteropolitanos, por estarem e ou serem de Salvador: Elvis Kempes e Davi Lemos, grandes amigos-irmãos, referências de estética humana a mim. Daniel Macêdo Batista e Joselino Rocha de Souza, amigos da maravilhosa Cidade Baixa de Salvador, rapazes de persistência, beleza d'alma, referência analítica, humildade e honestidade. Outros tantos amigos, colegas e professores da Universidade Federal da Bahia (UFBA), do Departamento de Ciência da Computação (DCC), do Bacharelado de Ciência da Computação, do Centro de Processamento de Dados (CPD): de quem não se esqueceu do apoio de vocês!

Saúdo os amigos, colegas e professores do Departamento de Automação e Sistemas (DAS) da Universidade Federal de Santa Catarina (UFSC). A secretaria do Programa de Pós-Graduação em Engenharia Elétrica. O ilustre Professor Joni da Silva Fraga por acolher-me em orientação, crítica científica e ensinamentos de vida. E, no mesmo patamar, o ilustre Professor (sim, Professor!) Alysson Neves Bessani, amigo querido, que, com sua co-orientação (oficial ainda que não expressa em capa), guiou-me responsavelmente nesta jornada: sua tenacidade e insistência e objetividade resultaram no que se escreve e descreve aqui. Todos os professores presentes na banca examinadora: Jean-Marie Farines (fantástico, Jean-Marie!), Magnos Martinello (quem, sem conhecer-me até a defesa, me acalmou com sãos elogios antes de defender o trabalho no dia 18 de outubro de 2006), Lau Cheuk Lung (quem me acordou silenciosamente com as questões sobre o outro lado da moeda do que apresentei) e Mario Dantas (simpática coincidência de parentesco!).

Saúdo Marcos Afonso, bahiano como eu, que me recebeu em Florianópolis naquele 22 de fevereiro de 2004, domingo de carnaval, recém-chegado de Salvador. As amigas bahianas Marizete e Luzia e o gaúcho-poeta-e-boêmio-e-pescador Marco Vasquez pelo incrível carinho quando eu mais precisava naquela moradia em quinteto (e mais além). Jomar e Márcia e o nosso arroz "biro-biro" com Guaraná Pureza (e quem o esquece?). A tríade parananese-interiorana: Edson, Marcos e Tinho, com quem me detive em um mesmo lar por duas semanas.

Saúdo os amigos de passagem, variados e distantes hoje: Andréia, Josiane, os amigos do GOU (Grupo de Oração Universitário); os amigos germânicos, distantes somente em meio físico, Vera e Sebastian; Delmo e Angela e Eduardo; Cassia Sigle, Clarice e Marina. Colegas do curso de Letras e Literatura Alemãs, professores dos departamentos de línguas vernáculas em estrangeira neste hum ano de convivência (maio a dezembro de 2006). Beth, Lucas, Jaque, Sandra, Clô e Lia, Maria; Aline, Beto e Camila, parceiros das aulas de dança; Dora e seu lindo óculos e corada meiguice. Maria

Lourdes e sua insustentável vontade que me fortificou e protegeu da maneira que pôde do tenebroso período de outubro e novembro de 2006. Alexandre, para quem volto-me com grande afeto e desejo de que seja feliz indefinidamente. Claudia e Gui e o avião de abril de 2004 quando tudo era saudade. Luciana Pacheco e os aviões diversos quando tudo era falta de dinheiro. Paulo Mafra e Jerusa Marchi. Rodrigo Sumar. Vilemar. Pessoal das baias dos doutorandos e do LIDAS: Fernando, Emerson, Eduardo, Rafael (Obelix), José Eduardo (Brandão), Cássia Tatibana, Pat. Anderson e Eliane. João Felipe e Davi. Kimie Nakahara, hoje bem longe, uma das primeiras amizades e parceira fértil nos trabalhos e ativismo com Software Livre. José Eduardo De Lucca, em quem sempre acredito e que merece o seu devido respeito. Djali e Delson Valois (Seu Delson). Achilles, Andreza, Andrea, Daiane, Ederson, Tatiane, Julia, Jonatas, Carlos e Rafael Savi. Aos colegas do GUFSC, Grupo de Usuários de Software Livre da UFSC, agora polarizados: Ricardo Grutz, Alberto, José, Fabio, Mathias, Christian. Douglas e Dionara, sempre carinhosos para comigo e dispostos em ajudar. Zacarias, sensível percepção dos dias, caribenha e parcimoniosa. Eda Fossati (Dona Eda), carinhos, mimos com conselhos sérios e para-todo-sempre, merendinhas e Guaranás Pureza em tamanho de criança. Ana Cristina, Luciene, Verinha, Natália, Fred(erico), Karina, João Paulo (JP), Fabio Favarim (Fabinho), Michelle Wangham (Micha) e Renato Donizetti (Renatinho). Saúdo a criançada da Casa São José e os dignos moradores da Serrinha, minha última morada neste período de mestrado.

Saúdo ao que se fez como grande companheiro, não somente um parceiro de moradia: Luiz Alcides Nascimento André, Luizinho. Guedes André (Seu Guedes, sempre presente!), Isabel Nascimento (Dona Isabel), Murilo, “Seo” Alcides, “Dona” Geysa e toda a família.

Saúdo a família Teles de Melo, a que se tornou a minha terceira família por conjunção e segunda de fato. Gilsélia Teles (Dona Gisélia), Aníbal Melo (Seu Aníbal), Samuel Melo. E quem me tomou inesperadamente e de forma tão natural e mantém-me constantemente em novas rotas, minha companheira: Noemi Teles de Melo.

Manifesto: Da Bizantinidade das Coisas (ou Da Importância de Ser Bizantino)

> Human nature is not a machine to be built after a model, and set to do exactly the work prescribed for it, but a tree, which requires to grow and develop itself on all sides, according to the tendency of the inward forces which make it a living thing.
> Such are the differences among human beings in their sources of pleasure, their susceptibilities of pain, and the operation on them of different physical and moral agencies, that unless there is a corresponding diversity in their modes of life, they neither obtain their fair share of happiness, nor grow up to the mental, moral, and aesthetic stature of which their nature is capable.
>
> (John Stuart Mill, *On Liberty*, 1859. Epígrafe mencionada no capítulo 4 – *The Economics of Social Production* – da obra *The Wealth of Networks* de Yochai Benkler, disponível em `http://www.congo-education.net/wealth-of-networks/`)

Este é um trabalho de Humanas. Este é um trabalho humanitário. Este é um trabalho com humanidade. Este é um trabalho com homens. Este é um trabalho de um homem.

Muito se passou durante esta jornada expressa em texto. Muito tempo.

E há muito gostaria de escrever este texto. O que me causa mais dor e mais prazer de toda esta dissertação. Nada do que foi escrito, do que está relacionado da próxima página em diante vale à pena. Apenas aqui.

Trata-se de um trabalho de só. De sóis.

Este é um manifesto de quem se despersonalizou durante a escrita de um texto insalubre. Desumano. Técnico. Metálico. Operacional. Movido pela força da raiva, pelo desejo de não escrever, de não descrever, de não obedecer a qualquer convenção ortográfica-técnica.

Em tempos de otimizar a personalização de serviços para pessoas mais aceleradamente impessoais – pegando de empréstimo uma fala de uma apresentação sobre gênero na UFSC (!) no segundo semestre de 2006, tão sofrido, que fechou (mas não terminou o fecho) com a apresentação deste trabalho – e de necessidades céleres de ótimos custo-benefício e de maior agilização de processos que se tornam progressivamente automatizados, vejo o outro, inefável: a degradação dos meios e intenções e manifestações humanas, a presença constante do Soldado-Amarelo (Graciliano Ramos, a querida professora Renata Telles do Centro de Comunicação e Expressão da UFSC, o querido professor Jair Fonseca, de Teoria da Literatura I, e os demais professores dos Departamentos de Vernáculas e Língua Estrangeira daquele mesmo Centro, com os quais tive um revolucionário contato entre maio de 2006 e dezembro do mesmo ano, bem como alguns dos mais próximos a mim no curso de Letras e Literaturas Alemãs sabem do que eu digo).

Vejo a triste expansão da insalubridade, da tragédia diária, da catástrofe silenciosa e ambiental (sua faceta, talvez, mais visível). Vejo a falta constante que apenas faz com que eu esteja em lugar

algum. São os propósitos de fazer-me escrever este texto sobre a importância de ser bizantino, tão combatido pelos rijos dizeres deste trabalho.

Por que diabos é importante ser bizantino?

A primeira resposta vem do conceito de bizantinidade à luz do que se entende em contexto computacional e que carrega um assombroso valor pós-computacional. Ser bizantino é não seguir o padrão dos outros cegamente, é permitir-se agir de maneira arbitrária, sua, na medida de não ter de obedecer a regra alguma de execução, de ação vinda de fora. É ser autônomo, auto-suficiente, dono de seu próprio pensamento, não seguir piamente padrões externos e externalizados; é não pensar em efetividade pelo valor literal e vulgarizado do verbete pelo jugo que não é seu. É guiar-se pelas sua criatividade e seu metabolismo, ainda que seja para receber o que os outros dizem. Porém, este recebimento vem de conveniência própria. Ser efetivo aos olhos do bizantino é desviar-se do desempenho absoluto que as cercanias esperam em favor da sua liberalidade relativa e do seu escrutínio. E ser consciente deste fato.

O significado de bizantino, a partir desta primeira impressão deve ser, então, reverberado. Estão nos manuais de artigo científico da área: no modelo de sistema, tipificado dos sistemas resistentes aos processos bizantinos, tudo o que se segue é estigmatizado, pressuposto. É o que fazem os processos corretos (com muitas aspas, faço ressaltar). E os bizantinos vêm para misturar os fatos, incomodar o congelado. Este ato de remexer, pertubar é fundamental, contudo. E, contudo, ele é fortemente combatido e, muitas vezes, combalido por tanto cansaço.

Neste cenário, convivi ao longo de toda a construção deste texto. Por detrás de cada palavra supostamente sólida, impávida, límpida, há um transgressor e, antes de tudo, subjetivo pensamento com vontade de jogar-se por toda a extensão das janelas, de lançar-se a fortes vontades próprias e indizíveis. Inexprimíveis por quaisquer formalizações presunçosas. Há uma vontade de ser bizantino.

Bizantinidade é fundamental. É fundamental cultivá-la. Cultivá-la conscienciosamente diante deste e outros tantos trabalhos que, por campanha de ilusão, primam em combatê-la com resistência e que, por este motivo, podem sugerir uma perigosa e definitiva coerência.

Florianópolis, Santa Catarina, 15 de Março de 2007, 21h40 (revisado em 04 de junho de 2007, 22h45).

Resumo da Dissertação apresentada à Universidade Federal de Santa Catarina como parte dos requisitos necessários para obtenção do grau de Mestre em Engenharia Elétrica.

Um Arcabouço de Avaliação de Algoritmos de Sistemas de Quóruns Bizantinos

Wagner Saback Dantas

Agosto/2006

Orientador: Joni da Silva Fraga
Área de Concentração: Controle, Automação e Informática Industrial
Palavras-chave: Sistemas Distribuídos, Tolerância a Faltas, Sistemas de Quóruns Bizantinos
Número de Páginas: xv + 105

A manutenção da disponibilidade e da integridade das informações é um requisito fundamental em sistemas de armazenamento de dados. Muitos destes sistemas devem manter estas propriedades mesmo em face à ocorrência de faltas acidentais ou intencionais (maliciosas), sendo que estas últimas são particularmente preocupantes uma vez que se originam de ataques bem sucedidos que levam a intrusões no sistema de armazenamento.

A fim de prover armazenamento que tolere faltas acidentais e maliciosas, podemos considerar que o sistema está sujeito a *faltas bizantinas* (a classe mais abrangente de faltas) e, então, empregar técnicas de tolerância a faltas bizantinas em sua concretização. Duas abordagens podem ser utilizadas para implementar sistemas de armazenamento tolerantes a faltas bizantinas: a Replicação Máquina de Estados e os *Sistemas de Quóruns Bizantinos*.

Sistemas de Quóruns Bizantinos (BQS) têm sido apresentados como uma boa abordagem para se construir armazenamento confiável distribuído, havendo muitas propostas para sua implementação. Escolher a melhor abordagem que satisfaça os requisitos de um ambiente de execução esperado exige uma avaliação minuciosa, que compreende o uso de ferramentas adequadas para modelagem e prototipação tanto do sistema de quóruns como do seu ambiente de execução. Apesar da boa quantidade de trabalhos sobre algoritmos de BQS, não existe uma ferramenta apropriada que viabilize um ambiente de testes para facilmente realizar tal tarefa de análise; ademais, não existem trabalhos que contemplem comparações e discussões entre os algoritmos propostos.

Esta dissertação tem como objetivo principal a implementação de um arcabouço de avaliação de algoritmos de Sistemas de Quóruns Bizantinos, denominado BQSNEKO. Para mostrar como este arcabouço pode ser usado para avaliação desta classe de algoritmos, o presente trabalho ainda apresenta e analisa casos de experimentos envolvendo algoritmos de BQS usando o próprio BQSNEKO, sobretudo em um ambiente de rede local. Estas análises, ao mesmo tempo em que comprovam a utilidade do BQSNEKO, servem como meio para discussão e um melhor entendimento dos algoritmos experimentados.

Abstract of Dissertation presented to Federal University of Santa Catarina as a partial fulfillment of the requirements for the degree of Master in Electrical Engineering.

A Framework for Evaluating Algorithms for Byzantine Quorum Systems

Wagner Saback Dantas

August/2006

Advisor: Joni da Silva Fraga
Area of Concentration: Control, Automation and Industrial Computing
Keywords: Distributed Systems, Fault Tolerance, Byzantine Quorum Systems
Number of Pages: xv + 105

Availability and consistency are essencial requirements of data storage systems. Most of such systems must maintain reliable and safe storage despite arbitrary faults (accidental or malicious faults). Malicious faults are particulary most critical than arbitrary ones since successful atacks may appear and cause intrusions on the storage system.

In these scenario of faults it is tipical to considerate systems subjected to *Byzantine Faults* (a broader class of faults) and to employ techniques of byzantine fault-tolerance. Two techniques can be used for implementing data storage systems with byzantine fault tolerate: the Replication State-Machine and the *Byzantine Quorum Systems*.

Byzantine Quorum Systems (BQS) have been presented like a good choice to build reliable distributed storage systems, existing many approaches for implementing them. Choosing the best approach that satisfies the requirements of an expected execution environment requires a careful evaluation which involves the use of adequate tools for modeling and prototyping both the Byzantine quorum system and its associated environment. However, to the best of our knowledge, there is no tool in which these task of analysis can be easily perfomed. In addiction there is no work that presents comparisons or that discusses the existing algorithms.

This dissertation mainly aims to present BQSNEKO, an useful framework for evaluating algorithms for Byzantine Quorum Systems. To show how BQSNEKO can be used for evaluating Byzantine quorum protocols, it will be described results of some experiments envolving algorithms for BQS using the BQSNEKO, especially as an execution environment of local network is regarded. At the moment such analyses show how useful the BQSNEKO is, they enable also discussions and a better understanding of the experimented algorithms.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

As crescentes exploração das redes de computadores e procura por soluções computacionais – muitas vezes, complexas – em diversos setores da sociedade, principalmente pelas soluções que se aproveitem do potencial de compartilhamento de recursos oferecido por estas redes, é palco para o grande interesse no uso e na implementação de aplicações computacionais distribuídas. Entretanto, ao mesmo tempo, este mesmo ambiente propício de rede detém um conjunto de características que se apresentam como um empecilho ao desenvolvimento dessa classe de aplicações. Trata-se de problemas que se expressam ora pela própria natureza heterogênea do ambiente de rede ligando computadores distintos em plataformas de *software* e *hardware*; ora pela sua incapacidade de servir informações importantes ao desenvolvimento de aplicações distribuídas, tal como uma referência global de tempo; ora pela possibilidade de falhas em componentes do sistema.

É neste cenário desafiador, motivado pelo potencial das redes, sobretudo pela expansão do uso da Internet, que a pesquisa em Sistemas Distribuídos se desenvolve. Ao lado das atividades de pesquisa que buscam novas soluções e tentam superar as dificuldades inerentes à área, o desenvolvimento de importantes serviços distribuídos (e.g., serviço de páginas *Web*) concorre para uma maior dependência de diversas aplicações distribuídas e, com efeito, para uma maior exigência de apreensão destas aplicações a atributos importantes para o seu bom funcionamento.

Em particular, um serviço de armazenamento distribuído representa uma parcela desta perspectiva de soluções potencializadas por uma rede de computadores. Em suma, neste tipo de serviço, cópias de um mesmo (conjunto de) dado(s) são mantidas em diferentes computadores espalhados pela rede, o que implica uma série de vantagens em relação a um serviço originalmente oferecido por um único servidor, tais como: *(a)* maior disponibilidade de dados aos usuários do serviço, uma vez que as informações estão em localidades distintas da rede e não em um ponto centralizado; *(b)* maior capacidade de resposta e *(c)* balanceamento de carga do serviço, pois um conjunto de servidores, utilizando um política apropriada de distribuição de requisições, conseguem atender um número maior de clientes à custa de menor carga individual do que um único servidor representando o serviço.

Porém, além da simples replicação dos sítios de dados, o que asseguraria a disponibilidade do sistema a priori, a construção de um serviço de armazenamento requer o uso de mecanismos adicionais

que o condicionem basicamente a: *(i)* suportar operações de atualização sobre suas cópias de dado a fim de preservar a sua propriedade de consistência sejam nos estados global (visão dos clientes em relação ao serviço como todo) e local (estado interno de cada servidor); *(ii)* manter a propriedade básica de disponibilidade ainda que ocorram falhas no sistema. Estas falhas podem surgir através de faltas acidentais ou intencionais (maliciosas), sendo que estas últimas são particularmente preocupantes uma vez que se originam de ataques bem sucedidos que levam a intrusões no sistema de armazenamento.

Sistemas de Quóruns Bizantinos (BQS, de *Byzantine Quorum Systems*) [32] são um meio de se prover consistência e disponibilidade em sistemas de armazenamento de dados replicados tolerantes a faltas acidentais e maliciosas. Assume-se neste caso que o sistema está sujeito a uma classe mais abrangente de faltas, isto é, a *faltas bizantinas* [26]. Nestes sistemas, os dados são replicados em diferentes conjuntos de servidores (quóruns) que compartilham servidores em comum. Assim, diferentes operações de leitura e escrita podem ser executadas em diferentes conjuntos de servidores, colaborando para a escalabilidade e o bom desempenho do sistema.

Na literatura, muitos algoritmos e abordagens para implementação de BQS já foram propostos (por exemplo, [27, 29, 32, 34, 35]). Estas soluções refletem diferentes perspectivas de projeto na construção de um sistema de armazenamento usando BQS a partir de certos aspectos chaves como o tamanho dos quóruns, o modelo de falhas dos clientes e a semântica de consistência suportada. A escolha de qual das abordagens seguir passa por uma avaliação minuciosa de qual algoritmo de BQS se adequa melhor ao ambiente esperado para a execução do sistema, o que demanda o uso de ferramentas apropriadas para construção e avaliação destes algoritmos. Até então, porém, não existe uma ferramenta que contemple tais tarefas.

1.1 Objetivos

Esta dissertação tem como objetivo geral apresentar a construção de um arcabouço para implementação e avaliação de sistemas de quóruns bizantinos, chamado doravante simplesmente de BQSNEKO. O BQSNEKO é um arcabouço desenvolvido sobre o simulador NEKO [47], sendo útil para análise de protocolos de sistemas de quóruns bizantinos. Aproveitando-se das funcionalidades providas pelo NEKO, o BQSNEKO permite a execução dos protocolos de BQS em redes simuladas ou reais considerando seus aspectos inerentes, como a ausência de tempo nos algoritmos e a simplicidade no lado servidor. A execução dos protocolos podem considerar cenários de ataques a partir da injeção de faltas bizantinas no sistema. O perfil de falta bizantina já pode ser oferecido pelo BQSNEKO ou possivelmente implementado usando facilidades oferecidas pelo arcabouço. A implementação de algoritmos para BQS torna-se muito mais simples usando o BQSNEKO, uma vez que várias tarefas necessárias para construção destes algoritmos já são suportadas pelo próprio arcabouço.

Como primeiro objetivo específico, a fim de demonstrar o uso do BQSNEKO, também são mostrados resultados de alguns experimentos envolvendo protocolos de BQS implementados no próprio arcabouço. Em princípio, os experimentos consistem na comparação de desempenho de operações

de leitura e escrita em diferentes sistemas de armazenamento utilizando os protocolos de BQS. De início, as configurações dos experimentos têm apenas protocolos de BQS, base de três das quatro situações de análise dos algoritmos. Em cada uma destas três situações de análise, dois algoritmos de BQS serão avaliados. O quarto caso de experimento envolve a comparação de desempenho também entre dois sistemas de armazenamento, porém um utilizará um algoritmo de BQS (desenvolvido no BQSNEKO), outro emprega uma técnica diferente para armazenamento bizantino com propriedades similares à primeira implementação (desenvolvida sobre o suporte de execução do NEKO sem o BQS-NEKO). O último caso de experimento tem como objetivo avaliar o desempenho de um algoritmo de BQS quando confrontado com outra técnica para implementação de armazenamento bizantino. Em todos os casos, à luz dos resultados obtidos, os testes indicam, ao mesmo tempo, alguns pontos fortes e fracos dos algoritmos e revelam situações em que um algoritmo é mais adequado para uso do que outro haja vista determinados ambientes de execução com variadas configurações de carga (concorrência de operações no sistema) e de falha bizantina nos servidores.

Sem perder o seu valor em contribuição, este trabalho tem como outro objetivo específico organizar, descrever e discutir os principais algoritmos de BQS propostos na literatura. Para tanto, em cada caso, se utiliza um formato padrão de apresentação – próprio deste texto – do funcionamento dos protocolos tanto em linguagem natural descritiva quanto em notação algorítmica. Ademais, o trabalho exibe e confronta as principais propriedades teóricas de cada algoritmo localizando ainda cada algoritmo apresentado com as suas principais contribuições no contexto científico.

1.2 Organização do Texto

A dissertação dispõe dos seguintes capítulos: o capítulo 2 apresenta uma visão geral sobre sistemas distribuídos e as principais técnicas de implementação de armazenamento distribuído tolerante a faltas bizantinas. O capítulo 3 apresenta o conceito de sistema de quóruns bizantinos, bem como descreve e discute os principais protocolos para sua implementação. O capítulo 4 descreve a arquitetura geral do NEKO e detalhada do arcabouço BQSNEKO, explicando as suas funcionalidades principais. Este capítulo mostra ainda um exemplo de como implementar um novo algoritmo para BQS, como construir um novo perfil de falta bizantina e como executar este novo protocolo com o perfil construído usando o BQSNEKO. O capítulo 5 descreve os casos de avaliação de algoritmos para BQS implementados no BQSNEKO, exibindo a partir dos seus resultados como o BQSNEKO pode ser usado para análise de algoritmos de BQS. O capítulo 6 acrescenta os últimos comentários acerca da dissertação e as conclusões finais do trabalho.

Capítulo 2

Sistemas Distribuídos e Conceitos Preliminares

2.1 Introdução

De maneira conceitual, um **sistema distribuído** se define como um sistema composto por computadores interligados em rede que, através de troca de mensagens, cumprem com um objetivo comum de execução. Estas entidades computacionais que constituem um sistema distribuído podem estar dispostas em um mesmo espaço físico ou espalhadas por localidades geográficas diferentes.

Por se aproveitarem das vantagens de uma rede de computadores, sistemas distribuídos são alvo de demanda crescente por parte da comunidade em geral. Tal fato se justifica pela natural capacidade de se compartilhar informação usando uma rede, sobretudo entre entidades usuárias a priori distintas (organizações, pessoas, computadores, etc.). Este cenário de troca de informação oportuniza um fértil campo para o desenvolvimento de serviços oferecidos por computador – e.g., serviços de armazenamento replicados pela rede –, potencializando o seu uso como ferramenta útil a diversos setores da sociedade.

Ao mesmo tempo, a mesma presença marcante e enriquecedora da infra-estrutura de rede, responsável pela criação de um ambiente para fomento de aplicações distribuídas, insere-se como um problema e uma limitação em sistemas distribuídos. Desta maneira, um sistema distribuído apresenta uma série de dificuldades adicionais, motivadas direta ou indiretamente pela atuação de uma rede de computador: problemas causados pela imprevisibilidade no tempo de transmissão de mensagens nos canais que comunicam entidades no sistema; problemas causados pela heterogeneidade de ambientes computacionais em *hardware* e *software*, com contextos de execução diferentes e pelas distâncias físicas entre pontos comunicantes; problemas ocasionados pela presença de falhas no sistema, que podem atingir qualquer um dos seus componentes e que, por isso, incrementam a complexidade de suas soluções.

De qualquer maneira, o fato é que a capacidade de integração e compartilhamento de recursos de uma rede possibilita às entidades computacionais de um sistema distribuído um ambiente apa-

rentemente comum, onde o conhecimento entre suas partes é, no mínimo, delimitado pela incerteza. Mesmo assim, cada uma destas mesmas partes independentes necessitam de uma execução que concorra para o progresso das finalidades do sistema computacional distribuído como um todo.

À luz desta situação-limite, onde a demanda por aplicações distribuídas progride e a sua existência recai em um ambiente de sistema natural e reconhecidamente problemático, surgem os desafios e o estímulo para a pesquisa e o desenvolvimento em Sistemas Distribuídos. Neste ínterim, muito já se produziu até então, de modelos primordiais de computação distribuída até aplicações avançadas, ora com resultados expressivos, ora com a descoberta de veredas ainda insolúveis ou inexploradas.

Este capítulo apresentará de maneira geral alguns destes resultados da pesquisa na área de Sistemas Distribuídos, especificamente os seus conceitos fundamentais no que diz respeito aos modelos mais conhecidos de computação distribuída e que se relacionam ao desenvolvimento deste trabalho. Portanto, os conceitos preliminares descritos aqui servirão como alicerce para todos os capítulos subseqüentes da dissertação.

Este capítulo se divide da seguinte forma: a seção 2.2 apresenta três visões de modelo de sistema distribuído – o modelo de comunicação (forma de interação entre os componentes do sistema) na seção 2.2.1; o modelo de tempo (hipóteses temporais dos componentes no sistema) na seção 2.2.2; e o modelo de falhas (como os componentes do sistema podem falhar) na seção 2.2.3. Tendo em vista a idéia de componentes falhos no sistema, a seção 2.3 aborda os aspectos que cercam o requisito de confiabilidade no funcionamento de sistemas distribuídos, enfatizando o uso de replicação em sistemas como meio de se implementar sistemas distribuídos confiáveis e resistentes a falhas (seção 2.3.1). Nesta seção, são descritas duas possíveis técnicas para sua concretização: a Replicação Máquinas de Estados [23, 41] (seção 2.3.1.1) e os Sistemas de Quóruns Bizantinos [32] (seção 2.3.1.2). Por se tratar de escopo específico desta dissertação, esta última técnica pode ser vista com maior detalhe no capítulo subseqüente. A seção 2.4 encerra com uma revisão do capítulo.

2.2 Modelos de Sistema

Um sistema distribuído pode ser imaginado e caracterizado através de alguns modelos fundamentais. Estes modelos descrevem o sistema a partir de determinados aspectos estruturais e comportamentais. A compreensão destes modelos é importante, uma vez que o desenvolvimento de aplicações distribuídas funcionais e condizentes com propriedades desejáveis de sistema (no todo ou em parte) dependem do conhecimento das abstrações expostas nos modelos computacionais distribuídos. As propriedades desejáveis de sistema pertencem a que se chama de **confiança no funcionamento (*dependability*)**, também conhecida como segurança de funcionamento ou "dependabilidade", e **segurança (*security)*** [2].

Os modelos de sistema descritos a seguir retratam três visões sobre um sistema distribuído no que tange à organização e à interação entre componentes do sistema (modelo de comunicação), às premissas de tempo dos componentes (modelo de tempo) e aos meios pelos quais os mesmos falham (modelo de falhas). Os modelos descritos tomam como base a idéia de um sistema distribuído por

passagem de mensagens (*message-passing*), onde entidades pertencentes ao sistema compartilham informações por troca de mensagens pela rede, ao invés de um sistema por **memória compartilhada (*shared-memory*)**, no qual estas mesmas entidades se comunicam usando objetos em memória compartilhada [21].

2.2.1 Modelo de Comunicação

Segundo [21], o modelo de comunicação (ou modelo de interação) aborda o sistema do ponto de vista estrutural, ou seja, relativo à organização e à descrição dos componentes do sistema e aos seus relacionamentos (como estes componentes interagem entre si). Dentre alguns modelos existentes, será falado apenas da comunicação ponto-a-ponto, modelo de interesse para este trabalho.

A comunicação ponto-a-ponto descreve a interação entre as entidades computacionais do sistema (**processos**) que se associam por meio de enlaces (**canais de comunicação**) que os ligam. Tal modelo pode ser descrito como um grafo, onde os vértices do grafo são processos, e as arestas, os enlaces.

Um enlace relaciona as primitivas de envio (*send*) e recebimento (*receive*) de mensagens entre dois processos comunicantes, denominados, por exemplo, de processos p e q. Os processos p e q possuem em sua estrutura local *buffers* de saída e de entrada. Então, quando p envia uma mensagem m para q, p põe m no *buffer* de saída (envio), o enlace que os liga transporta m até o *buffer* de entrada de q (transporte), onde m é retirada e recebida por q (entrega).

Cada processo executa uma série de passos ou operações pré-concebidas pelo **algoritmo distribuído** a depender do seu papel no sistema (por exemplo, processos com perfis fixos de cliente e servidor numa arquitetura cliente-servidor, ou processos com perfis iguais em uma arquitetura par-a-par ou *peer-to-peer*). Um algoritmo distribuído estabelece o conjunto de instruções a serem realizadas por cada processo participante do sistema, incluindo as suas possíveis trocas de mensagens. Cada perfil de processo pode ser representado formalmente por um respectivo autômato de execução, que consistiria em um conjunto de estados possíveis para um processo naquele perfil (incluindo estados inicial e final) e um conjunto de transições entre estes estados associado aos possíveis eventos no sistema.

2.2.2 Modelo de Tempo

O modelo temporal descreve o sistema distribuído haja vista as hipóteses de tempo sobre os seus componentes principais (processos e canais de comunicação). Dentre o conjunto de modelos de tempo encontrados na literatura, dois se destacam e se situam em dois extremos opostos: o modelo síncrono (com fortes premissas de tempo) e o modelo assíncrono (com fracas premissas de tempo). Em suma, estes dois modelos especificam o comportamento temporal de um sistema distribuído em três pontos:

1. O tempo para realização dos passos de processamento de uma entidade participante do sistema (processo) haja vista o seu autômato de execução;

2. A taxa de atualização nos relógios locais dos processos do sistema. Os relógios locais são responsáveis por mapear eventos incidentes no processo (e.g., envio e recepção de mensagens) em tempo [23]. A diferença das taxas de atualização de relógios dos processos em um sistema revela duas propriedades importantes dos relógios: a **precisão externa ou *accuracy***, que é o grau de desvio dos relógios em relação a um tempo de uma referência fora do sistema; e a **precisão interna**, que é a diferença entre dois relógios de dois processos de um determinado sistema;

3. O atraso total de uma mensagem, que se define como a soma dos tempos de envio, transporte e entrega da mensagem.

O **modelo síncrono** define um limite conhecido de tempo para os passos de execução em um processo, para o atraso de uma mensagem e para o limite máximo na precisão interna ou externa. Imaginar um sistema sobre este modelo nem sempre é possível, por exemplo, para certos tipos de rede (e.g., em redes de larga escala como a Internet), uma vez que, nestes casos, não se consegue estimar valores precisos para limite de tempo na entrega de mensagens. Entretanto, ainda que o modelo síncrono não sirva para casos como este, o mesmo pode ser útil em outras situações como modelo para estudo do funcionamento de algoritmos distribuídos.

Por outro lado, o **modelo assíncrono** não estabelece limites conhecidos de tempo para a execução de passos em um processo, para as precisões interna ou externa e para o atraso de mensagens. Este modelo mais simples, por não impor restrições de tempo, representa um comportamento mais verossímil de um sistema distribuído, onde as velocidades relativas dos processos são variadas e as latências dos canais de comunicação possuem a rigor limites imprevisíveis. Por outro lado, a incerteza de tempo deste modelo dificulta a construção de aplicações distribuídas na prática, já que a garantia de término na execução de um algoritmo distribuído depende de uma previsibilidade nas interações entre processos.

Ao lado dos modelos síncrono e assíncrono, existem modelos intermediários denominados **modelos de sincronia parcial**. Estes modelos podem definir premissas de tempo relacionadas a algum dos componentes do sistema (execução de processos, latência de entrega de mensagens e desvio de relógios) e se transformam em alternativas úteis para representar a maioria dos sistemas na prática.

2.2.3 Modelo de Falhas

Descreve os pressupostos de falhas nos componentes do sistema distribuído, projetando de que forma processos e canais de comunicação podem se desviar de seus comportamentos previamente especificados (**comportamento correto**) e, desta maneira, apresentarem um **comportamento faltoso**. A descrição de um modelo de falhas é fundamental porque propicia a construção de meios para se contornar a possibilidade de falhas no sistema distribuído, tornando-o mais propenso a contemplar propriedades desejáveis, tal como propriedades relacionadas à confiabilidade de funcionamento.

Hadzilacos e Toueg estabeleceram em [21] uma classificação conceitual de falhas para processos e enlaces de comunicação. Estes desvios no sistema são agrupados em falhas por omissão, falhas temporais e falhas arbitrárias.

As **falhas por omissão (*omission failures*)** podem ocorrer tanto em processos quanto em canais de comunicação: em processos, acontecem por **parada de funcionamento (*crashing*)**, por **falha de omissão no envio (*send-omission*)** – processo emissor considera que enviou a mensagem, mas não a coloca no seu *buffer* de saída – ou omissão na recepção (*receive-omisson*) – processo receptor tem uma mensagem em seu *buffer* de entrada, mas não a recebe; em canais, acontece por **omissão no enlace (*channel-omission*)** – processo emissor coloca mensagem em seu *buffer* de entrada, mas a mensagem não chega ao *buffer* de entrada do processo receptor.

As **falhas temporais (*timing failures*)** [14] incidem em sistemas com alguma hipótese de tempo (sistemas em modelos completamente síncronos ou em modelos de sincronia parcial). Em relação a processos, as falhas temporais podem ser por **falha de relógios (*clock failure*)** – em algum momento no sistema, há algum desvio nos relógios locais dos processos em relação ao tempo de referência estabelecido – ou por **falha de desempenho (*perfomance failure*)**, onde processos não realizam seus passos de execução dentro do limite de tempo previsto; em canais, esta mesma falha de desempenho acontece quando os enlaces não transmitem mensagems dentro do limite de tempo previsto.

As **falhas arbitrárias (*arbitrary failures*)** representam um tipo mais genérico e complexo de falhas. Falhas arbitrárias são também chamadas de **falhas bizantinas ou *Byzantine failures***, em alusão ao problema descrito por Lamport et al. em [26] denominado *Problema dos Generais Bizantinos*, que ilustra situações de falha onde componentes de um sistema computacional (na alegoria do problema, membros de um grupo de generais do exército bizantino) podem oferecer informações conflitantes a outros diferentes componentes causando o mal funcionamento do sistema. Neste modelo de falhas, processos encenam qualquer tipo de falha, de maneira acidental e benigna ou intencional e maliciosa. Processos podem, por exemplo, parar de funcionar total ou parcialmente ou continuar funcionando modificando valores de mensagens e enviando aos processos corretos. Esta mesma idéia pode ser estendida para canais, que podem apresentar falhas arbitrárias duplicando mensagens transmitidas, enviado-as em ordens diferentes ou modificando o valor das mensagens.

2.3 Segurança de Funcionamento de Sistemas Distribuídos

A hipótese de ocorrência de falhas nos serviços de um sistema (representada pelo modelo de falhas) e a preocupação de que aplicações distribuídas devem cumprir com requisitos de **confiança no funcionamento (*dependability*)** [2, 13] acarretam o aparecimento e o desenvolvimento de meios adequados para se concretizar sistemas confiáveis. Neste contexto, temos, como base, algumas propriedades desejáveis de sistema de acordo com a confiança no funcionamento:

- **Confiabilidade (*Reliability*)**: continuidade do serviço correto do sistema;
- **Segurança (*Safety*)**: ausência de conseqüências catastróficas sobre os usuários do sistema;

- **Reparabilidade (*Manutenability*)**: capacidade do sistema de receber modificações e reparos;
- **Disponibilidade (*Availability*)**: prontidão do serviço correto do sistema;
- **Integridade (*Integrity*)**: ausência de alterações inadequadas no sistema.

Ao lado das propriedades desejáveis ao sistema, existem as **ameaças à confiabilidade de funcionamento [2, 13]**. Estas ameaças, representadas pelo modelo de falhas no sistema (seção 2.2.3), se traduzem pela possibilidade de ocorrência de faltas, erros e falhas. Uma **falta** é a causa remota de uma falha à medida que a presença e a ativação daquela habilita mais tarde a incidência desta. Uma falta pode ser motivada tanto em períodos de desenvolvimento quanto de operação ou manuseio de sistemas por entidades externas que interagem com o sistema (outros sistemas computacionais, seres humanos, meio ambiente, etc.) ou por entidades internas ao sistema (**componentes**). Entende-se aqui por componente quaisquer entidades de um sistema que interagem entre si por meio de seus estados externos (interfaces de serviço). Um componente, quando considerado não atômico, se traduz também em um sistema formado por outros componentes. Por exemplo, um processo é um componente do sistema distribuído e, ao mesmo tempo, um sistema formado por componentes de *hardware* e *software*.

Um **erro** se define como um desvio de comportamento provocado por uma falta que se reflete no estado interno de um componente do sistema. Caso esta manifestação seja imediata, chamamos de um erro causado por uma *falta ativa*; caso contrário, trata-se de um erro causado por uma *falta dormente*. Quando um erro se propaga de um estado interno de um componente para um estado externo do sistema, atingindo o seu serviço oferecido, temos uma **falha**.

Assim, por exemplo, quando uma falta ocorre em um sistema s, esta falta, de início, alcançou um certo componente c_s de s, podendo se refletir ou não como um desvio de comportamento (erro), que muda o estado interno de c_s. Em caso de ativação da falta, tem-se a possibilidade de o erro em c_s poder gerar uma falha no sistema s, desde que este erro se manifeste em um estado externo (serviço) de s. Usando este mesmo raciocínio, podemos considerar o mesmo componente c_s como um sistema s', que falha pela ativação de uma falta em seu componente $c'_{s'}$, gerando o erro deste último e a propagação como falha em s'; esta falha no sistema s' é equivalente à falta no componente c_s (lembrando que s' e c_s são a mesma entidade), ativando o seu erro e, por fim, causando a falha no sistema s. Ou, de outra forma: falta (em $c'_{s'}$) $\xrightarrow{ativa}$ erro $\xrightarrow{propaga}$ falha (em c_s) $\xrightarrow{causa}$ falta (em c_s) $\xrightarrow{ativa}$ erro $\xrightarrow{propaga}$ falha (em s).

Dado o conjunto de ameaças em aplicações distribuídas, um conjunto adequado de técnicas e métodos deve ser adotado com o objetivo de assegurar as suas propriedades de confiabilidade no funcionamento. Estes procedimentos podem ser classificados da seguinte forma [2, 13]:

- **Prevenção de Faltas (*Fault Prevention*)**: oferece meios para se prevenir a ocorrência de faltas. Agrega as tarefas corriqueiras de metodologia de projeto em *software* e *hardware*;
- **Tolerância a Faltas (*Fault Tolerance*)**: previne o aparecimento de falhas pressupondo a presença de faltas no sistema;

- **Supressão de Faltas (*Fault Removal*)**: atua na diminuição do número ou da severidade de faltas do sistema usando técnicas de verificação e validação em sua etapa de projeto para *software* ou *hardware*;

- **Previsão de Faltas (*Fault Forecasting*)**: emprega técnicas de modelagem e teste para estimar o número e as conseqüências de faltas futuras no sistema.

2.3.1 Usando Replicação para Tolerância a Faltas

Uma maneira de se implementar Tolerância a Faltas no sistema é através do seu **Mascaramento de Faltas (*Fault Masking*)**. Para tanto, é comumente empregada a técnica de **Replicação** de *hardware* ou *software*: distribuir a cópia de um certo serviço da aplicação (código e dados) para outros servidores como forma de mantê-lo funcionando ainda que ocorram falhas em um número suficiente de servidores no sistema. Este número "suficiente" é conhecido conceitualmente como o *número máximo de faltas (f-threshold)* f do serviço implementado, embora alguns autores utilizem também a notação t para designar o limite de faltas. Em outras palavras, f significa o número máximo de servidores faltosos que o sistema consegue suportar para continuar funcionando corretamente [1]. O número n de servidores (réplicas) do sistema é definido em função deste f. Usando replicação entre processos, sabe-se que o número n de réplicas no sistema deve ser de, pelo menos, $f+1$ servidores para que o sistema tolere f faltas por parada.

No que se refere ao *serviço de armazenamento distribuído de dados*, duas técnicas podem ser utilizadas para implementar Replicação visando à Tolerância a Faltas:

- *Máquina de Estados* [23, 41]: técnica *geral* de construção de implementações tolerantes a faltas para qualquer serviço determinista, onde a execução de cada operação esteja condicionada à realização de outras operações casualmente precedentes. Esta técnica se fundamenta na utilização de protocolos de acordo para garantir que todas as réplicas do serviço executem o mesmo conjunto de operações em uma mesma ordem;

- Os *Sistemas de Quóruns* [19, 42]: técnica *específica* para implementação de serviço de armazenamento distribuído através da execução de leituras e escritas de dados em diferentes conjuntos de servidores (quóruns) que mantêm réplicas (servidores) em comum. Este trabalho apenas considera a possibilidade de faltas por parada nos servidores. Uma extensão deste trabalho, com a hipótese de faltas bizantinas nos servidores, foi concebida por Malkhi e Reiter em [32]: é o que se chama de *Sistemas de Quóruns Bizantinos*.

De maneira comum às duas técnicas, um serviço abstrato de armazenamento distribuído é modelado em uma *arquitetura cliente-servidor*, na qual um servidor mantém um *registrador* [24] r que suporta operações $r.write(v)$ (escrever um valor v em r) e $r.read()$ (devolver o valor atual do registrador r). Estas operações são invocadas remotamente por processos clientes nos registradores

[1] Como bem comenta [41], um sistema que é tolerante a f faltas pode continuar a executar corretamente, porém tal funcionamento correto não é garantido.

implementados nos servidores. Os servidores corretos não permitem a atualização dos registradores de outra forma que não seja seguindo os protocolos definidos.

2.3.1.1 Replicação Máquina de Estados

A Replicação Máquina de Estados [23, 41] é o método mais empregado na concretização de sistemas distribuídos tolerantes a faltas. A implementação deste serviço genérico de replicação requer **determinismo de réplicas**: partindo de um mesmo estado inicial e após executarem a mesma sequência de operações, as réplicas devem ter a mesma evolução percorrendo os mesmos estados intermediários até alcançarem o mesmo estado final [41].

O requisito de determinismo de réplicas exige que o sistema tenha as seguintes propriedades: *(i.)* difusão confiável com ordem total das requisições (todas as réplicas executam o mesmo conjunto de requisições e na mesma ordem); *(ii.)* os estados das réplicas são alterados apenas pela execução de requisições; *(iii.)* as operações executadas nas réplicas devem ser deterministas (em qualquer réplica, a execução da mesma seqüência de operações a partir de um mesmo estado inicial leva a um mesmo estado final). As propriedades *(ii.)* e *(iii.)* são garantidas diretamente pelas propriedades do sistema: um registrador é, por definição, determinista, e os servidores implementados ignoram qualquer pedido de alteração do valor do registrador que não seja enviado seguindo o protocolo de escrita definido. Desta forma, é preciso definir um protocolo para difusão com ordem total ou difusão atômica para satisfazer a propriedade *(i.)*. O ponto central para concepção deste protocolo reside na resolução do *Problema de Consenso*, já que a resolução da difusão atômica e do consenso são equivalentes, isto é, resolver um dos dois problemas automaticamente resolve o outro, e vice-versa [12].

Em um sistema distribuído composto por diversos processos independentes, o **Problema do Consenso** consiste em fazer com que todos os processos corretos acabem por decidir (Terminação) o mesmo valor (Acordo), o qual deve ter sido previamente proposto por algum dos processos do sistema (Validade[2]). Entretanto, sabe-se que, em ambientes assíncronos, não é possível a resolução do consenso de maneira determinista mesmo que canais sejam confiáveis e haja, pelo menos, um processo que falhe por parada (**impossibilidade FLP**) [17]. Esta comprovação teórica é de suma importância para a área de Sistemas Distribuídos tendo como efeito o aparecimento de algumas soluções que se valem de algum modelo especializado do modelo assíncrono para contorná-la. Assim, é possível obter acordo entre processos em ambiente assíncrono usando, por exemplo, *Algoritmos Aleatórios de Consenso* [7, 8, 38] (solução probabilista) ou *Detectores de Falha* [12] (solução que encapsula as propriedades de tempo da rede em um módulo ligado a cada processo do sistema, responsável pela manutenção de uma lista de processos faltosos). Outro resultado teórico importante é a constatação de que, por ser equivalente ao consenso, a difusão atômica está restrita também à impossibilidade FLP [12]. Já [47] afirma que, na prática, se considerássemos redes locais, a impossibilidade FLP não surtiria efeito (i.e., algoritmo de consenso termina).

A fim de implementar a propriedade *(i.)* para obter o determinismo do modelo de replicação máquina de estados, considerando a premissa de até f faltas bizantinas no sistema, é possível encon-

[2]Esta é apenas uma das possíveis definições da propriedade de Validade.

trar alguns trabalhos com soluções para o consenso: com soluções probabilistas, por exemplo, em Ben-Or [7] (em meio assíncrono usando $5f+1$ servidores), em Rabin [38] (usando $4f+1$ servidores em meio síncrono com canais confiáveis e $10f+1$ servidores em meio assíncrono), em Bracha e Toueg [8] (em meio assíncrono usando $3f+1$ servidores) e em Malkhi e Reiter [28] ($3f+1$ servidores em meio assíncrono); com soluções deterministas, por exemplo, em Zielinski [49] ($3f+1$ servidores em meio parcialmente síncrono) e em Martin et al. [33] ($5f+1$ servidores em meio assíncrono). Outros trabalhos apresentam soluções para a difusão atômica suportando faltas bizantinas, como Reiter em [40] (considerando sistemas assíncronos usando $3f+1$ servidores) e Castro e Liskov em [11] (em meio assíncrono usando $3f+1$ servidores).

A seção 5.3.4 apresenta uma comparação entre dois sistemas de armazenamento tolerantes a faltas bizantinas, que implementam, respectivamente, uma abordagem para replicação máquinas de estado e outra para sistemas de quóruns. No caso da primeira abordagem, o protocolo de consenso empregado é o PAXOS Bizantino [11, 25], que, nesta situação, adiciona modificações em sua implementação apresentadas em [33, 49]. Doravante, este protocolo será chamado apenas de PAXOS. A fim de esclarecer o seu funcionamento, daremos aqui uma visão geral do funcionamento do protocolo PAXOS.

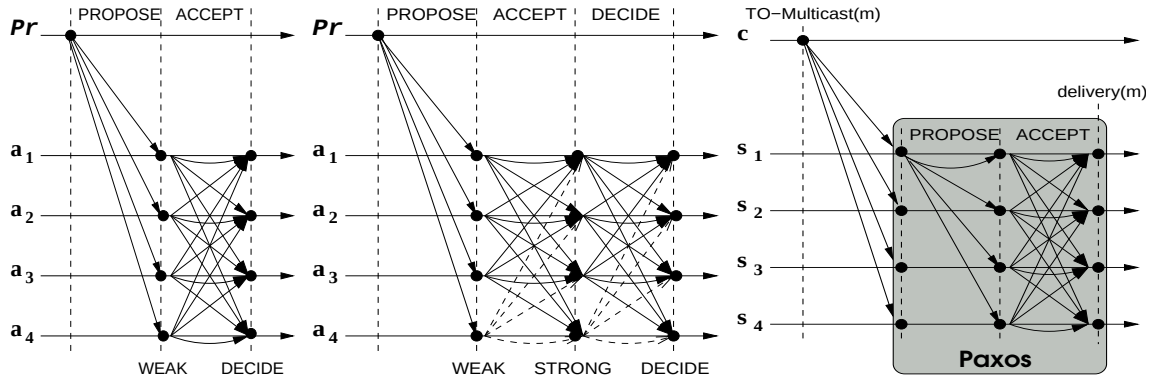
O algoritmo PAXOS considera três classes de agentes: **proponentes**, os quais propõem os valores; **aceitantes**, os quais escolhem um único valor entre os propostos; **aprendizes**, os quais precisam aprender o valor decidido. Em nossa implementação, todos os servidores do sistema desempenham estes três papéis ao mesmo tempo; no entanto, distinções serão feitas para facilitar o entendimento do protocolo.

Este algoritmo é executado em *rounds*, sendo que, em cada *round r*, um proponente p_r é escolhido líder. Este líder tem a responsabilidade de escolher e enviar uma proposta aos aceitantes, os quais tentarão fazer deste valor proposto a decisão do consenso através de uma ou mais fases de trocas de mensagens visando garantir o Acordo. Por fim, quando estabelecida, a decisão de consenso é enviada aos aprendizes. As propriedades de segurança sempre são mantidas pelo protocolo, mas o consenso somente terá progresso em *rounds* favoráveis. Um *round* é considerado *favorável* quando seu líder é correto (cada round tem apenas um líder, o processo $r\%n$, onde n é o número de réplicas no sistema) e o sistema está num período de sincronia: as comunicações e computações ocorrem dentro de um período de tempo limitado. Nesta situação, um valor proposto pode ser aprendido dentro do período de um *round*. Adicionalmente, um *round* é dito *muito favorável* se ele é favorável e não existem falhas nos aceitantes. Caso um *round r* não seja favorável, um novo *round* é iniciado com um novo líder e assim sucessivamente até que um valor seja aprendido.

As Figuras 2.1(a) e 2.1(b) ilustram alguns cenários de execução do PAXOS Bizantino A Figura 2.1(a) mostra uma execução em que o protocolo executa um *round* muito favorável e consegue terminar em apenas dois passos. Note que, como os próprios aceitantes são os aprendizes, não é necessário difundir o valor decidido para estes últimos. Este padrão segue as otimizações definidas em [33, 49]. O caso normal de operação do PAXOS, onde o primeiro *round* é favorável, é apresentado na Figura 2.1(b): um *round* do algoritmo consolida uma decisão em três passos de comunicação [11]. Caso um *round* não seja completado em um determinado intervalo de tempo, um novo *round* é iniciado através de um protocolo de transição, que requer dois passos de comunicação. Este protocolo de troca é o

único passo do protocolo onde criptografia de chave pública é necessária, para garantir a autenticidade do próximo valor a ser proposto pelo líder recém-escolhido; logo, o protocolo não requer este mecanismo de criptografia em execuções favoráveis.

A implementação da difusão com ordem total usando o PAXOS se baseia na execução de uma instância deste algoritmo para cada mensagem a ser ordenada. Desta forma, uma requisição *m* é a *i*-ésima requisição a ser executada se e somente se for o resultado da execução *i* do PAXOS [11, 33]. A Figura 2.1(c) ilustra a difusão com ordem total de uma requisição usando o PAXOS em uma execução muito favorável. Nesta figura não consideramos o envio das respostas pelos servidores para o cliente.



(a) Favorável. (b) Favorável com a_4 faltoso. (c) Difusão com ordem total.

Figura 2.1: Execuções do PAXOS.

Uma otimização usualmente implementada em replicação máquina de estados é a tentativa de execução de algumas operações sem a necessidade de execução do protocolo de ordem total. Com esta otimização, toda operação que não altere o estado do serviço (uma leitura, por exemplo) é enviada aos servidores, que respondem imediatamente. Se o cliente obtém $n - f$ respostas iguais, a operação termina; caso contrário, a requisição é reenviada através da difusão com ordem total. Esta otimização permite que uma leitura seja completada em dois passos de comunicação (envio e resposta) em ocasiões onde não existem faltas ou operações de escrita sendo executadas concorrentemente. A implementação da operação de leitura de nosso serviço de armazenamento usa essa otimização.

Como consideramos faltas bizantinas, um líder (proponente) malicioso pode propor uma operação inexistente para execução, ferindo a vivacidade do sistema e fazendo com que os servidores fiquem bloqueados esperando o recebimento desta requisição. A resolução deste problema é descrita em [11]: o líder deve ser trocado, e requisições inexistentes já ordenadas devem ser definidas como operações *nop*, que não alteram o estado do sistema.

2.3.1.2 Sistemas de Quóruns Bizantinos

Sistemas de quóruns bizantinos [32] implementam sistemas replicados de armazenamento de dados distribuídos com garantias de consistência e disponibilidade mesmo com a ocorrência de faltas

bizantinas em algumas de suas réplicas. Ao contrário dos sistemas baseados no modelo máquinas de estados, um protocolo para sistemas de quóruns não exige a execução de acordo entre as réplicas para seqüenciamento das operações, o que livra esta solução da impossibilidade FLP [17] e permite sua implementação com terminação garantida em sistemas assíncronos. Algoritmos para sistemas de quóruns são reconhecidos por seus bons desempenho e escalabilidade, já que os clientes desse sistema acessam de fato somente um conjunto particular de servidores ao invés de todos os servidores.

Servidores em um sistema de quóruns organizam-se em subconjuntos denominados **quóruns**. Cada dois quóruns de um sistema mantém um número suficiente de servidores corretos em comum (garantia de consistência), sendo que existe pelo menos um quórum no sistema formado somente por servidores corretos (garantia de disponibilidade) [32]. Os clientes realizam suas operações em registradores de leitura e escrita replicados por estes quóruns, cujos tamanhos para operações de leitura e escrita podem ser iguais (**quóruns simétricos**) ou não (**quóruns assimétricos**). Cada registrador detém um par $\langle v,t \rangle$ com um valor v do dado armazenado e uma estampilha de tempo (*timestamp*) t associada. Este *timestamp* é definido pelo cliente quando de sua operação de escrita, sendo que cada cliente c utiliza conjuntos disjuntos de *timestamps*.

Na literatura de sistema de quóruns bizantinos, muitas construções e protocolos para sistemas de quóruns têm sido propostos (por exemplo, [27, 32, 34]). Estas construções se diferenciam basicamente pelas premissas de faltas assumidas nos clientes do sistema (bizantinos ou não), pelo número de servidores no sistema (em função do limite de f faltas bizantinas nestes servidores) e pela natureza simétrica ou assimétrica dos quóruns de leitura e escrita.

2.4 Revisão do capítulo

Este capítulo apresentou alguns conceitos e modelos fundamentais estudados em Sistemas Distribuídos e que estão relacionados ao desenvolvimento desta dissertação. Dentro do contexto do modelo de faltas, apresentou-se a idéia de usar replicação como um procedimento comum para tolerar possíveis ocorrências de falhas no sistema. Em especial, introduziram-se duas técnicas conhecidas para construção de serviço de armazenamento distribuído de dados tolerante a faltas bizantinas. Primeiramente, o modelo Máquina de Estados; depois, os Sistemas de Quóruns Bizantinos. No modelo Máquina de Estados, foi explicado o funcionamento do algoritmo PAXOS, que voltará mais adiante no capítulo 5 em um caso de avaliação com um algoritmo para sistemas de quóruns. Já os Sistemas de Quóruns Bizantinos estão no escopo principal deste trabalho e, por este motivo, terá um espaço reservado nesta dissertação. O próximo capítulo será dedicado a este assunto.

Capítulo 3

Algoritmos de Sistemas de Quóruns Bizantinos

3.1 Introdução

Sistemas de quóruns bizantinos (BQS, de *Byzantine quorum systems*) [32] são um meio de se implementar consistência de dados e disponibilidade de serviço em sistemas replicados de armazenamento mesmo com a ocorrência de faltas bizantinas [26]. Este conceito decorre do clássico conceito de sistemas de quóruns [19, 42], onde se admite somente processos que apresentam falhas por parada (*crashing*). Ao contrário de sistemas de armazenamento baseados no paradigma de Máquinas de Estado [23, 41], a implementação de BQS não requer acordo entre as réplicas que implementam o serviço em relação ao mesmo conjunto de operações a serem executadas e à ordem das mesmas. Por este motivo, os BQS não são suscetíveis à impossibilidade FLP [17]. Outros pontos que favorecem o uso dos BQS são o bom desempenho e a boa escalabilidade apresentados pelos seus algoritmos, uma vez que os clientes do sistema acessam efetivamente somente um quórum de servidores ao invés de todos os servidores. Entretanto, BQS possuem a limitação de somente implementar armazenamento que suporte operações de leitura e escrita.

Nos BQS, objetos de memória compartilhada (**registradores**) são emulados em um ambiente distribuído usando passagem de mensagens. Processos servidores se organizam em conjuntos de subconjuntos de servidores chamados **quóruns**, onde, para quaisquer dois quóruns, existe um número suficiente de servidores corretos em sua interseção (garantia de consistência). Além disso, num sistema de quóruns, pelo menos, um quórum é formado apenas por servidores corretos (garantia de disponibilidade). Cada processo servidor é um repositório de dados que armazena uma cópia local de um registrador que suporta acessos para leitura e escrita. Clientes realizam operações nestes registradores por meio de **protocolos de sistemas de quóruns**.

Processos nos BQS estão sujeitos a falhas bizantinas, podendo apresentar um comportamento que foge do especificado no seu algoritmo e executar qualquer tipo de ação (maliciosa ou não) no

sistema. Até f processos servidores podem ser faltosos (bizantinos). Alguns algoritmos também toleram clientes faltosos.

3.1.1 Objetivo

Neste capítulo, são apresentados o conceito e a descrição dos principais protocolos de acesso para alguns sistemas de quóruns bizantinos encontrados na literatura. Para cada sistema de quóruns apresentado, são descritos os algoritmos de leitura e escrita dos clientes, o algoritmo implementado pelos servidores, as características dos quóruns utilizados e dos dados envolvidos nas operações de acesso aos servidores. Em todos os casos, até um limite f de servidores caracterizam-se por falhas de natureza bizantina. Clientes, em algumas situações indicadas, podem ser bizantinos.

3.1.2 Organização do capítulo

Este capítulo organiza-se da seguinte forma: a seção 3.2 fala dos conceitos básicos de um sistema de quóruns bizantinos, apresentando o modelo de sistema e os vários tipos de registradores emulados por sistema de quórum. Esta seção também caracteriza os BQS encontrados na literatura. As seções de 3.4 a 3.6 descrevem os algoritmos dos clientes e dos servidores para diferentes configurações de BQS em relação ao modelo de falha dos clientes e ao tipo de quórum empregado (tabela 3.1). Para cada algoritmo apresentado, são retratadas também a sua complexidade de mensagens (número de mensagens geradas pelo algoritmo) e a quantidade de passos de comunicação necessários para realizá-lo. Cada subtópico destas seções (conteúdo da tabela 3.1) identifica os algoritmos apresentados pelas suas semânticas de consistência. A seção 3.7 exibe um resumo de todas as características dos sistemas de quóruns apresentados e apresenta um histórico até então dos trabalhos relacionados. A seção 3.8 dispõe as considerações finais.

	Simétrico	**Assimétrico**	**Mínimo**
Clientes corretos	3.4.1.1 [32] 3.4.1.2 [29] 3.4.1.3 [29]	3.5.1.1 [35] 3.5.1.2 [35]	3.6.1.1 [34] 3.6.1.2 [34]
Clientes faltosos	3.4.2.1 [32] 3.4.2.2 [29] 3.4.2.3 [27]	—	3.6.2.1 [34] 3.6.2.2 [34]

Tabela 3.1: Modelo de falhas dos clientes por construções de quóruns bizantinos

3.2 Conceitos básicos

3.2.1 Modelo do sistema

O sistema considerado (definido em [32]) consiste em dois conjuntos de processos: servidores no conjunto U, onde $|U| = n$, e clientes num conjunto Π arbitrário, distinto de U e possivelmente

infinito. Um **sistema de quórum** $\mathcal{Q}$ é um conjunto não vazio de subconjuntos de U ($\mathcal{Q} \subseteq 2^U$), onde $\forall Q_1, Q_2 \in \mathcal{Q}$, $Q_1 \cap Q_2 \neq \emptyset$. Processos participam de um sistema assíncrono e interagem através de canais ponto a ponto confiáveis e autenticados.

Processos no ambiente distribuído podem apresentar falhas bizantinas [26] seja, por exemplo, por perda de mensagens, parada total do processo ou executando passos imprevistos em sua especificação de funcionamento. No caso de concretização de falhas, servidores e clientes são denominados **faltosos** ou **bizantinos**, caso contrário, são vistos como **corretos**. Neste ambiente, até f servidores podem falhar. O valor f é denominado o **limite de faltas ou *f-threshold***.

O trabalho original de Malkhi e Reiter [32] desenvolve uma idéia mais geral em relação ao limite de faltas, introduzindo o conceito de **sistema passível de falhas (*fail-prone system*)** $\mathcal{B}$. Compreende-se tal conjunto $\mathcal{B}$ como um conjunto não vazio de subconjuntos de U ($\mathcal{B} \subseteq 2^U$), onde $\forall B_1, B_2 \in \mathcal{B}$, $B_1 \nsubseteq B_2$. Um $B \in \mathcal{B}$ representa um conjunto de servidores faltosos, caracterizando cenários de falhas que podem ocorrer, nos quais até f servidores podem falhar. A variável f é definida como em relação ao conjunto $\mathcal{B}$ como $|B| \leq f$. Ao longo da descrição dos algoritmos, para facilitar o seu entendimento, usaremos apenas o limiar de faltas.

Cada quórum $Q \in \mathcal{Q}$ é acessado por clientes em operações de leitura e escrita. Um **quórum de leitura (Q_r)** é aquele acessado pelos clientes em operações de leitura. Um **quórum de escrita (Q_w)** é aquele acessado pelos clientes em operações de escrita. Quando $|Q_r| = |Q_w|$, temos um sistema de **quóruns simétricos**. Caso contrário, temos um sistema de **quóruns assimétricos**.

3.2.2 Registradores

Em suas operações de leitura e escrita, processos clientes acessam uma variável x armazenada em um registrador replicado em um conjunto U de servidores. A variável x, conforme definido em [32], é um par $\langle v,t \rangle$, onde v é o valor da variável e t é sua estampilha de tempo (*timestamp*) associada, criada quando da escrita do valor v. Para que cada par $\langle v,t \rangle$ gerado seja único, cada cliente c detém um subconjunto T_c de *timestamps*, tal que $\forall c_1, c_2 \in \Pi$, $T_{c_1} \cap T_{c_2} = \emptyset$. Cada $t_{c_1} \in T_{c_1}$ pode ser formado, por exemplo, concatenando, nesta ordem, um número de série local do cliente que gerou o *timestamp*, representando o momento em que o valor v foi criado, e o seu identificador único.

Os dados envolvidos nas operações de acesso ao servidor podem ser genéricos (*generic data*) ou auto-verificáveis (*self-verifying data*). **Dados auto-verificáveis** são aqueles submetidos a algum processo de assinatura criptográfica cujo valor resultante permite detectar se as informações contidas neste foram modificadas ou não após sua escrita. Isto é útil, por exemplo, nos casos de verificação da integridade de algum valor num ambiente onde clientes corretos assinam valores com uma chave privada secreta, escrevem nos registradores replicados os valores assinados e detectam, por uma chave pública correspondente, na leitura, se os valores foram modificados por servidores bizantinos. Já os **dados genéricos**, ao contrário dos dados auto-verificáveis, não são digitalmente assinados.

Registradores podem apresentar diferentes níveis de acesso, suportando, por exemplo, a escrita de apenas um único cliente por vez, caracterizando uma **semântica "único escritor" (*single-writer***

***semantic*)**, ou de vários clientes escritores, acarretando uma **semântica "vários escritores" (*multi-writer semantic*)**. No caso do acesso às variáveis compartilhadas com leituras e escritas concorrentes, o comportamento do registrador dependerá de sua semântica de consistência. Segundo Lamport [24], existem três tipos de semânticas de consistência (em ordem crescente de força), que definem o comportamento de uma variável compartilhada numa situação de concorrência: **segura (*safe*), regular (*regular*) e atômica (*atomic*)**:

- **Semântica segura**: garante que, se não houver escrita concorrente, uma operação de leitura devolve o valor da última escrita realizada, caso contrário (leitura com escrita concorrente), a leitura pode resultar em qualquer valor do domínio de valores do registrador;
- **Semântica regular**: garante a semântica segura e, quando houver escritas concorrentes à leitura, o valor lido pode ser o da última escrita realizada ou um dos valores sendo escritos;
- **Semântica atômica**: garante operações de leitura e escrita dentro de uma semântica regular e de uma ordenação de leituras e escritas segundo a relação *happens before* ($\rightarrow$) [1] [23]. Assim, uma leitura r retorna o valor escrito pela última operação de escrita realizada w, ou seja, $w \rightarrow r$, sendo que não há outra operação de escrita w' tal que $(w \rightarrow w') \wedge (w' \rightarrow r)$.

3.2.3 Sistema de quóruns bizantinos

Um sistema de quóruns bizantinos garante os requisitos de disponibilidade e consistência de armazenamento mesmo na presença de servidores que podem falhar de maneira arbitrária (bizantina). Um sistema de quóruns bizantinos apresenta subconjuntos de servidores em quóruns Q com uma quantidade de servidores corretos suficiente, tal que a propriedade de consistência do sistema seja mantida mesmo com a ocorrência de faltas bizantinas.

Um exemplo de sistema de quóruns deste tipo (sistema Q) é mostrado na figura 3.1. Tal sistema compreende um subsistema passível de falhas $\mathcal{B}$, um conjunto de quóruns $\{Q_1, Q_2, Q_3\}$ e os subconjuntos $B \in \mathcal{B}$. A consistência em Q é garantida pela interseção entre cada dois quóruns no conjunto $\{Q_1, Q_2, Q_3\}$, sendo que cada interseção contém uma quantidade suficiente de servidores corretos (por exemplo, o conjunto $Q_1 \cap Q_2 \setminus B$). Logo, se um cliente realizar duas operações em dois quóruns diferentes, necessariamente um mesmo grupo de servidores corretos será acessado, ainda que servidores bizantinos sejam possivelmente acessados também (conjunto $B \cap Q_2$). A disponibilidade é garantida pela existência de, pelo menos, um quórum $Q \in Q$, onde todos os servidores são corretos (conjunto Q_3).

A seguir, são apresentados os vários tipos de BQS estudados neste trabalho.

[1] Se a e b são eventos, então $a \rightarrow b$ se a acontece antes de b num mesmo processo, ou se a é o evento de escrita em um processo e b é o evento de leitura correspondente em outro processo, ou se, dado um evento c, $a \rightarrow c$ se $a \rightarrow b$ e $b \rightarrow c$

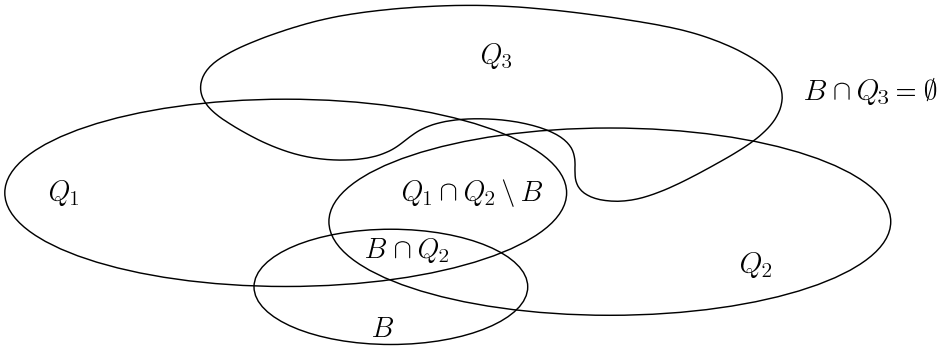


Figura 3.1: Representação formal de um sistema de quóruns bizantinos.

3.2.3.1 Sistemas f-mascaramento [32]

Um **sistema de quórum de f-mascaramento (*f-masking quorum system*)** pressupõe o armazenamento de dados genéricos usando quóruns simétricos. O sistema organiza-se de maneira que, na interseção entre cada dois quóruns, há uma maioria de servidores corretos. Desta maneira, para f servidores bizantinos, há em cada interseção pelo menos $2f+1$ servidores, garatindo, ainda no pior caso, uma maioria de $f+1$ servidores corretos. Assim, considerando quaisquer dois quóruns Q_1 e Q_2 neste sistema, temos:

$$|Q_1 \cap Q_2| \geq 2f+1$$

Se quisermos aplicar uma interseção de $2f+1$ entre Q_1 e Q_2, ambos de tamanho $|Q|$, limitando-se a um conjunto total de *n* servidores, teremos ao final o tamanho de cada quórum:

$$|Q_1|+|Q_2|-n \geq 2f+1 \Rightarrow 2|Q|-n \geq 2f+1 \Rightarrow |Q| \geq \tfrac{2f+1+n}{2} \Rightarrow |Q| = \lceil \tfrac{2f+1+n}{2} \rceil$$

Sabendo que existe, pelo menos, um quórum com todos os servidores corretos ($|Q| \leq n-f$), temos:

$$|Q| \leq n-f \Rightarrow \tfrac{2f+1+n}{2} \leq |Q| \leq n-f \Rightarrow \tfrac{2f+1+n}{2} \leq n-f \Rightarrow n \geq 4f+1$$

Assim, temos um sistema com um número de servidores de $n \geq 4f+1$, o que permite a construção de quóruns com no mínimo $3f+1$ servidores.

3.2.3.2 Sistemas f-disseminação [32]

Um **sistema de quórum de f-disseminação (*f-dissemination quorum system*)** possui construção similar ao f-mascaramento. A diferença aqui está no armazenamento de dados auto-verificáveis, o que enfraquece a premissa de interseção entre cada dois quóruns. Neste caso, somente se exige que

haja pelo menos 1 servidor correto na interseção, cujo valor pode ser verificado como correto. Logo, para um sistema com f servidores bizantinos, temos uma interseção de $f+1$ servidores, ou seja, se Q_1 e Q_2, temos $|Q_1 \cap Q_2| \geq f+1$.

Utilizando o mesmo raciocínio dos quóruns de f-mascaramento, obtemos um sistema com quóruns simétricos de tamanho $|Q| = \lceil \frac{n+f+1}{2} \rceil$. Com f servidores a menos em cada interseção entre dois quóruns, é possível ter um sistema com $n \geq 3f+1$ sevidores, o que permite a construção de quóruns com no mínimo $2f+1$ servidores.

3.2.3.3 Sistemas a-mascaramento [35]

Assim como nos quóruns de f-mascaramento, um **sistema de quórum de a-mascaramento (*a-masking quorum system*)** armazena dados não assinados (genéricos). Logo, se restringe que, na organização do quórum, exista pelo menos $2f+1$ servidores na interseção entre um quórum qualquer de leitura com um quórum qualquer de escrita.

Entretanto, ao contrário dos quóruns de f-mascaramento, temos quóruns de leitura e escrita com tamanhos diferentes (assimétricos), sendo que os quóruns de escrita não cumprem com o requisito de disponibilidade, apenas os quóruns de leitura. A disponibilidade para os quóruns de escrita não é necessária pois suas operações não esperam por mensagens de confirmação para serem concluídas (escritas não confirmáveis). Nesta situação, para que haja pelo menos um quórum de leitura com apenas servidores corretos, os quóruns de escrita têm de ser maiores que os quóruns de leitura em f servidores. Ou seja, $|Q_w| = |Q_r| + f$. A partir desta constatação e seguindo o raciocínio aplicado para o quórum de f-mascaramento, temos:

$$|Q_r| + |Q_w| - n \geq 2f+1 \Rightarrow |Q_w| - f + |Q_w| - n \geq 2f+1 \Rightarrow 2|Q_w| \geq 2f+1+n+f \Rightarrow |Q_w| \geq \frac{2f+1+n+f}{2} \Rightarrow |Q_w| = \lceil \frac{n+f+1}{2} \rceil + f$$

Logo, $|Q_r| = \lceil \frac{n+f+1}{2} \rceil$. Sabendo que $|Q_r|$ respeita a propriedade de disponibilidade ($|Q_r| \leq n-f$), temos:

$$|Q_r| \leq n-f \Rightarrow \frac{n+f+1}{2} \leq |Q_r| \leq n-f \Rightarrow \frac{n+f+1}{2} \leq n-f \Rightarrow n \geq 3f+1$$

Assim, temos um sistema com $n \geq 3f+1$ servidores. No caso do menor número possível de servidores no sistema ($3f+1$), temos quóruns de leitura e escrita, respectivamente, com tamanhos de $2f+1$ e $3f+1$.

3.2.3.4 Sistemas a-disseminação [35]

Assim como os quóruns de f-disseminação, um **sistema de quórum de a-disseminação (*a-dissemination quorum system*)** armazena dados auto-verificáveis, acarretando um arranjo de sistema

onde existem pelo menos $f+1$ servidores na interseção entre cada dois quóruns. E, tal como os quóruns de a-mascaramento, um quórum de a-disseminação pressupõe quóruns assimétricos, onde a propriedade de disponibilidade é somente respeitada para o quórum de leitura.

Partindo dessas premissas e combinando os raciocínios usados na definição dos quóruns de f-disseminação e a-mascaramento, obtemos os seguintes resultados: tamanho do quórum de leitura $|Q_r| = \lceil \frac{n+1}{2} \rceil$, tamanho do quórum de escrita $|Q_w| = \lceil \frac{n+1}{2} \rceil + f$ e quantidade de servidores no sistema $n \geq 2f+1$ servidores, o que permite a formação de quóruns de leitura e escrita, respectivamente, com tamanhos mínimos de $f+1$ e $2f+1$.

3.2.3.5 Sistemas "mínimos" [34]

O que se considera aqui como um **sistema de quórum "mínimo"** é em princípio um sistema de quórum de a-mascaramento à medida que ambos armazenam dados genéricos, organizam-se em quóruns assimétricos, mantêm pelo menos $2f+1$ servidores em comum na interseção entre cada par de quóruns de leitura e escrita e, finalmente, possuem $n \geq 3f+1$ servidores.

Entretanto, no que se refere à organização dos quóruns, existem diferenças entre os quóruns mínimos e os quóruns assimétricos. Ao contrário do sistema de a-mascaramento, onde se assegura a propriedade de disponibilidade aos quóruns de leitura, em um sistema de quórum mínimo, apenas os quóruns de escrita parecem manter tal propriedade. Desta maneira, pode-se pensar em um raciocínio próximo do que foi mostrado no sistema de a-mascaramento obtendo uma inversão nos tamanhos dos quóruns de leitura e escrita em comparação ao que se estabeleceu nos sistemas de a-mascaramento: $|Q_r| = \lceil \frac{n+f+1}{2} \rceil + f$ (ou $|Q_r| = \lceil \frac{n+3f+1}{2} \rceil$ conforme [34]) e $|Q_w| = \lceil \frac{n+f+1}{2} \rceil$.

Observação: de fato, a palavra "mínimo", usada aqui para nomear o sistema ora apresentado, diferenciando-o do sistema de a-mascaramento, referencia-se ao termo *minimal* empregado por Martin et. al em [34]. Neste trabalho, *minimal* diz respeito ao limite mínimo para se construir sistemas de quóruns bizantinos. Em tal caso, para se tolerar f faltas, é necessário, no mínimo, $3f+1$ servidores para que se obtenha um sistema com qualquer semântica de consistência e com suporte a escritas confirmáveis; por outro lado, para se construir um sistema com escritas não confirmáveis, são necessários $2f+1$ servidores. Ainda assim, o termo *minimal* pode se referir mesmo aos próprios sistemas de quóruns que empreguem estes número mínimos de servidores.

3.2.4 Notação e funções básicas

Além das notações apresentadas nas seções anteriores, de agora em diante, outras notações de variáveis e funções serão usadas neste capítulo na descrição dos algoritmos. São elas:

- S: um conjunto (letra em maiúscula);
- s: um elemento (letra em minúscula);

- $s[\,]$ ou $S[\,]$: um vetor;
- $min\{S\}$: função que devolve um elemento com valor mínimo dentre todos os elementos pertencentes a um conjunto numérico ou a um domínio de valores qualquer S com ordem pré-estabelecida;
- $max\{S\}$: função que devolve um elemento com valor máximo dentre todos os elementos pertencentes ao conjunto numérico ou a um domínio de valores qualquer S com ordem pré-estabelecida;
- $\langle \text{tipo}[,dados]\rangle$: mensagem de um tipo que contém possivelmente alguns *dados*;
- $\langle i\rangle_p$: informação i (mensagem ou dado) assinada por um processo p;
- *proof*: certificado do par $\langle v_s,t_s\rangle$ armazenado em um servidor s. Dependendo do algoritmo empregado, $proof$ pode ser assinado pelo cliente que escreveu o par em s, pelo próprio servidor s ou, ainda, pode ser um conjunto de certificados (e.g., mensagens assinadas). O conjunto $(\langle v_s,t_s\rangle, proof)$ corresponde a um dado armazenado em s juntamente com o seu certificado $proof$;
- $sign\_ok(\langle i\rangle_p)$: função booleana que verifica se uma informação i (mensagem ou dado) assinada por um processo p é autêntica, ou seja, se é de fato uma informação do processo p não modificada por outro processo;
- $valid(i,C)$: função booleana que verifica se uma informação i é válida de acordo com um certificado (ou um conjunto de certificados) C;
- *last_ts*: o último *timestamp* calculado por um cliente. Iniciado com valor zero;
- $\#_{elem}S$: número de elementos iguais a *elem* no vetor (ou conjunto) S;
- S_{elem}: subconjunto (ou subvetor) do conjunto (ou vetor) S cujos elementos são iguais a *elem*;
- $send(p,msg)$: primitiva de comunicação que envia uma mensagem *msg* para um processo p;
- $receive(p,msg)$: primitiva de comunicação bloqueante para recepção de uma mensagem *msg* de um processo p.

Algoritmo 1 Função de consulta em um quórum (dados sem assinaturas)

function $query(q)$
1: $\forall s \in U$, $send(s,\langle \text{QUERY}\rangle)$
2: $S[1\ldots n] \leftarrow \perp$
3: **repeat**
4: **wait** $receive(s,\langle \text{QUERY-RESPONSE}, \langle v_s,t_s\rangle\rangle)$
5: $S[s] \leftarrow \langle v_s,t_s\rangle$
6: **until** $\#_{\perp}S = n-q$
7: **return** S
end function

A função *query* (algoritmo 1), executada por um cliente c, consulta pares $\langle v,t\rangle$ armazenados em um quórum, passando como parâmetro o tamanho deste quórum (argumento q). Ela funciona do

seguinte modo: através de uma mensagem QUERY, um cliente requisita um conjunto de pares $\langle v,t \rangle$ para todos os servidores e espera por um conjunto de tamanho q contendo respostas de servidores. Em seguida, armazena os pares no vetor S (inicialmente vazio). Em cada posição s de S ($S[s]$), estão o *timestamp* t_s (conjunto $S[\,].ts$) e o valor v_s (conjunto $S[\,].v$), retornados por um servidor s (linhas 1 a 6). A função devolve para o cliente este vetor S ao final (linha 7).

Algoritmo 2 Função de consulta em um quórum (dados com assinaturas)

```
function query_w_sign(q)
 1: ∀s ∈ U, send(s, ⟨QUERY⟩)
 2: S[1...n] ← ⊥
 3: repeat
 4:    wait receive(s, ⟨QUERY-RESPONSE, ⟨v_s, t_s⟩, proof⟩)
 5:    if valid(⟨v_s, t_s⟩, proof) then
 6:       S[s] ← ⟨v_s, t_s⟩
 7:    end if
 8: until #_⊥ S = n − q
 9: return S
end function
```

A função *query_w_sign* (algoritmo 2) funciona de maneira quase idêntica à função *query* (algoritmo 1). Porém, neste caso, cada servidor s devolve numa mensagem QUERY-RESPONSE, além do seu par armazenado $\langle v_s,t_s \rangle$, um certificado *proof* deste par. O par recebido só é aceito se estiver de acordo com certificado *proof* (linhas 5 a 7). Porém, em alguns algoritmos, a consulta pode devolver também pares cujas assinaturas sejam inválidas, omitindo a execução da linha 5, quando o cliente não possui a chave pública para verificação do par recebido. Esta função é usada somente nos casos de sistemas que armazenam dados auto-verificáveis.

3.3 Estrutura geral dos algoritmos de BQS

Os protocolos de leitura e escrita de BQS seguem algumas concepções gerais, o que nos permite tratá-los de uma maneira simplificada, sem considerar seus pontos mais singulares, tais como a organização dos quóruns e as propriedades de consistência envolvidas no armazenamento das réplicas. Com o objetivo de introduzir a descrição dos algoritmos de BQS, esta seção apresenta uma visão geral de como funcionam a leitura e a escrita em tais sistemas, o que facilitará na compreensão dos algoritmos mais específicos mostrados na seções subseqüentes.

A idéia de uma operação de leitura compreende estruturalmente os seguintes passos de execução (figura 3.2): **(a) passo de consulta**, ou seja, usando uma mensagem QUERY, o cliente consulta inicialmente os pares $\langle v,t \rangle$ armazenados em um quórum Q, cujos servidores respondem através de uma mensagem QUERY-RESPONSE; **(b) passo pós-consulta**, onde, com o conjunto de pares do quórum, o cliente escolhe qual par será devolvido pela operação (normalmente, o par com maior *timestamp*) e, ao final, realiza alguma ação (operação SOME_ACTION); **(c) passo de devolução**, onde o cliente, finalmente, devolve um valor v. Como se percebe, é o passo (b) que caracteriza um protocolo de leitura e, ao mesmo tempo, se relaciona à semântica de consistência de um algoritmo.

Já a concepção de um algoritmo de escrita consta em geral dos seguintes passos (figura 3.3): **(a) cálculo de *timestamp***, que envolve, de início, um possível passo de consulta a um quórum de leitura (conforme visto na descrição genérica da operação de leitura). O cliente gera o novo *timestamp* para o novo valor que deseja escrever em um quórum. Quando não há um consulta explícita a um quórum, o cliente utiliza a sua informação local para calcular um novo *timestamp*. Em seguida, há o **(b) passo de escrita**, onde o cliente escreve em todos os servidores[2] um novo par $\langle v', t' \rangle$, onde t' corresponde ao *timestamp* calculado no passo (a). Este par é enviado numa mensagem UPDATE. Por fim, o cliente participa opcionalnente do **(c) passo de confirmação** ao esperar por mensagens de confirmação de um quórum de escrita.

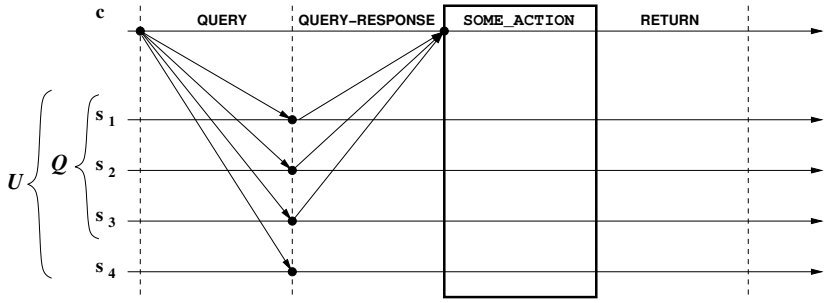


Figura 3.2: Funcionamento geral dos algoritmos de leitura de BQS

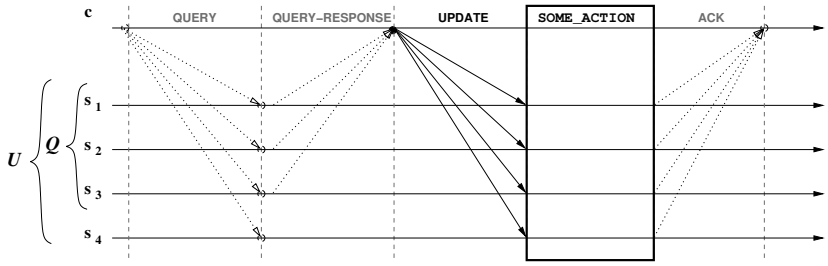


Figura 3.3: Funcionamento geral dos algoritmos de escrita de BQS

3.4 Algoritmos para sistemas de quóruns simétricos

Esta seção descreve os algoritmos de armazenamento em sistema de quóruns simétricos, ou seja, com quóruns de leitura e escrita de mesmo tamanho.

3.4.1 Clientes corretos

Esta seção descreve os algoritmos de BQS de quóruns simétricos que não toleram clientes faltosos.

[2] Ou, conforme descrito para alguns algoritmos, em um quórum de escrita.

3.4.1.1 MWMR seguro [32]

Este caso descreve algoritmos de leitura e escrita em sistema de quóruns de f-mascaramento (vide seção 3.2.3.1). A semântica de consistência alcançada é *multi-writer multi-reader* segura.

Funcionamento da escrita (algoritmo 3). O algoritmo de escrita (procedimento *write*) possui duas fases. **Fase 1**: cliente c requisita um conjunto de pares $\langle v,t\rangle$ a um quórum Q usando a função *query* (ver algoritmo 1 na seção 3.2.4) (linha 1). Depois de receber todos os pares de Q, c define o seu menor valor de *timestamp* t, que é maior que todos os *timestamps* recebidos de Q (linhas 2 e 3). **Fase 2**: cliente prepara mensagem UPDATE com o novo par $\langle v,t\rangle$, a envia para todos os servidores e espera um conjunto de confirmações de um quórum (linha 5).

Algoritmo 3 Escrita de um cliente c

procedure *write*(v)
1: $S \leftarrow query(|Q|)$
2: $max\_ts \leftarrow max\{S[\,].ts\}$
3: $t \leftarrow min\{t_c \in T_c : max\_ts < t_c\}$
4: $\forall s \in U$, $send(s, \langle \mathsf{UPDATE}\ , \langle v,t\rangle\rangle)$
5: **wait** $receive(q, \langle \mathsf{ACK}\rangle)$, $\forall q \in Q'$
end procedure

Funcionamento da leitura (algoritmo 4). O algoritmo de leitura (função *read*) possui apenas uma fase. Na linha 1, o cliente c requisita um conjunto de pares armazenados $\langle v,t\rangle$ ao quórum Q e os armazena no vetor S (inicialmente vazio) usando a função *query* (algoritmo 1 na seção 3.2.4); caso o cliente encontre em S um conjunto de $f+1$ pares iguais $\langle v,t\rangle$, este devolve $\langle v,t\rangle$ (linha 3). Caso contrário, devolve $\perp$, ou seja, um valor vazio indicando falha na leitura (linha 5).

Algoritmo 4 Leitura de um cliente c

value **function** *read*$()$
1: $S \leftarrow query(|Q|)$
2: **if** $\exists\langle v,t\rangle : \#_{\langle v,t\rangle} S \geq f+1$ **then**
3: **return** v
4: **else**
5: **return** $\perp$
6: **end if**
end function

Execução do servidor (algoritmo 5). O algoritmo do servidor é bem simples, uma vez que não se consideram clientes faltosos no sistema. Quando o servidor recebe uma mensagem QUERY, ele devolve o valor e o *timestamp* associado armazenados em seu registrador para o cliente. Quando o servidor recebe a mensagem UPDATE, ele atualiza seu estado se o *timestamp* recebido t for maior que o seu *timestamp* armazenado t_s. Independentemente de esta condição ser satisfeita, o servidor envia uma mensagem de confirmação para o cliente.

Complexidade de mensagens: os algoritmos MWMR seguro executam com complexidade de troca de mensagens de $O(n)$. As operações de escrita e leitura realizam-se, respectivamente, em 4 e 2 passos de comunicação.

Algoritmo 5 Execução de um servidor s

upon $receive(c, \langle \mathsf{QUERY} \rangle)$

1: $send(c, \langle \mathsf{QUERY\text{-}RESPONSE}, \langle v_s, t_s \rangle \rangle)$

upon $receive(c, \langle \mathsf{UPDATE}, \langle v, t \rangle \rangle)$

1: **if** $t > t_s$ **then**
2: $\quad \langle v_s, t_s \rangle \leftarrow \langle v, t \rangle$
3: **end if**
4: $send(c, \langle \mathsf{ACK} \rangle)$

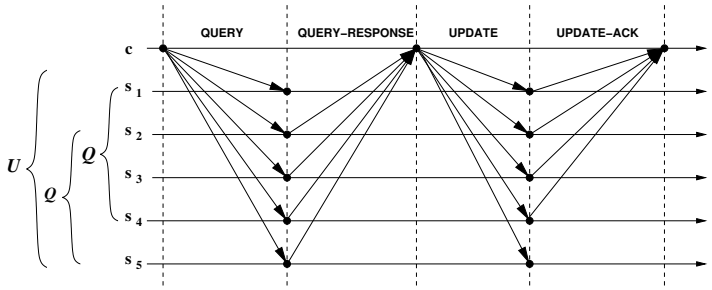


Figura 3.4: Protocolo de escrita – quóruns simétricos, clientes corretos e MWMR seguro para $f = 1$

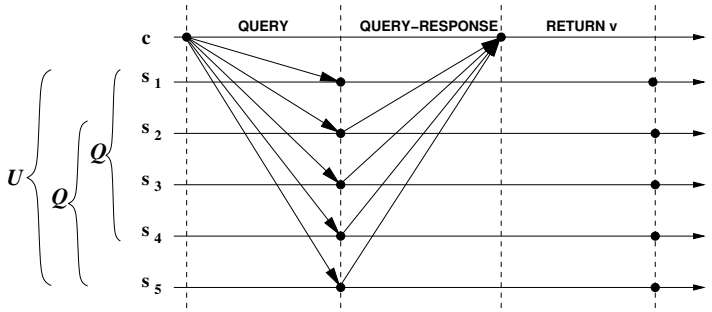


Figura 3.5: Protocolo de leitura – quóruns simétricos, clientes corretos e MWMR seguro para $f = 1$

3.4.1.2 MWMR regular [32]

Este caso descreve algoritmos de leitura e escrita em sistema de quóruns de f-disseminação (seção 3.2.3.2). A semântica de consistência alcançada é *multi-writer multi-reader* regular. Os clientes corretos são responsáveis pela assinatura dos dados armazenados, o que garante uma semântica regular.

Para cada cliente, é usado um par de chaves criptográficas. A chave privada de um cliente é usada na assinatura de suas informações nas operações de escrita em um quórum. A chave pública é usada para verificação das informações assinadas por aquele cliente nas operações de leitura dos clientes. Note que este algoritmo requer que os clientes conheçam as chaves públicas uns dos outros, o que pode ser um empecilho à escalabilidade do sistema.

Funcionamento da escrita. A escrita neste caso é muito parecida ao procedimento *write* na seção 3.4.1.1 (algoritmo 3). A única diferença existente aqui é a escrita de dados auto-verificáveis (o par $\langle v, t \rangle$ escrito é assinado pelo cliente) ao invés dos dados não assinados do caso MWMR seguro.

Funcionamento da leitura (algoritmo 6). A leitura (função *read*) realiza-se em uma fase e é similar ao algoritmo de leitura do caso MWMR seguro, exceto pela verificação dos pares recebidos do quórum (garantia de semântica regular). O cliente requisita um conjunto de pares auto-verificáveis válidos de um quórum Q e os armazena no vetor S (inicialmente vazio) usando a função *query_w_sign* (algoritmo 2). Depois de receber todos os pares do quórum Q, o cliente seleciona o par com maior *timestamp*.

Algoritmo 6 Leitura de um cliente c

value **function** *read*()
1: $S \leftarrow query\_w\_sign(|Q|)$
2: $max\_ts \leftarrow max\{S[\,].ts\}$
3: **return** v'
end function

Execução do servidor (algoritmo 7). A execução do servidor assemelha-se ao protocolo do servidor na seção 3.4.1.1 (algoritmo 5). Quando o servidor recebe uma mensagem QUERY, devolve o seu dado armazenado, que é auto-verificável. Quando o servidor recebe uma mensagem UPDATE de um cliente c, este atualiza o seu estado somente se: o par assinado estiver corretamente assinado pelo cliente c, e se o *timestamp* t contido nesta mensagem for maior que o *timestamp* t_s já armazenado.

Algoritmo 7 Execução de um servidor s

$\{c' \in \Pi$, tal que c' escreveu anteriormente $\langle v_s, t_s \rangle$ no servidor $s\}$
upon $receive(c, \langle \mathsf{QUERY} \rangle)$
1: $send(c, \langle \mathsf{QUERY\text{-}RESPONSE}, \langle v_s, t_s \rangle_{c'} \rangle)$
upon $receive(c, \langle \mathsf{UPDATE}, \langle v, t \rangle_c \rangle)$
1: **if** $sign\_ok(\langle v, t \rangle_c)$ **then**
2: **if** $t > t_s$ **then**
3: $\langle v_s, t_s \rangle \leftarrow \langle v, t \rangle$
4: **end if**
5: **end if**
6: $send(c, \langle \mathsf{ACK} \rangle)$

Complexidade de mensagens: os algoritmos MWMR regular executam com complexidade de troca de mensagens de $O(n)$. As operações de escrita e leitura realizam-se, respectivamente, em 4 e 2 passos de comunicação.

3.4.1.3 MWMR atômico [29]

Este caso é similar ao visto na seção 3.4.1.2, exceto pelo fato de a semântica de consistência alcançada ser *multi-writer multi-reader* atômica.

Funcionamento da escrita. A escrita neste caso é idêntica à escrita vista no caso MWMR regular (seção anterior).

Funcionamento da leitura (algoritmo 8). A leitura neste caso possui 2 fases. Na sua segunda fase, uma operação de reescrita (*write back*) garante que o valor lido por um cliente seja lido por todos os outros clientes leitores antes da próxima escrita, implicando uma semântica atômica deste

protocolo. **Fase 1**: idem à fase 1 do protocolo de leitura na seção 3.4.1.2 (algoritmo 6). **Fase 2**: cliente c envia WRITE-BACK com par $\langle v', t' \rangle$ assinado (linha 6), espera um conjunto de confirmações de um quórum e retorna o valor v'. Uma otimização neste passo (citada em [32]) é a realização da reescrita apenas para os servidores que não responderam a consulta com o par assinado $\langle v', t' \rangle$.

Algoritmo 8 Leitura de um cliente c

value **function** *read*()
1: $S \leftarrow query\_w\_sign(|Q|)$
2: $max\_ts \leftarrow max\{S[\,].ts\}$
3: $\{c' \in \Pi$, tal que c' escreveu anteriormente $\langle v_s, t_s \rangle$ em um servidor correto $s \in U\}$
4: $\langle v', t' \rangle \leftarrow \{\langle v_s, t_s \rangle_{c'} \in S : t_s = max\_ts\}$
5: $S' \leftarrow \{\forall s' : S[s'] = \langle v', t' \rangle\}$
6: $\forall s \in U \setminus S', send(s, \langle$WRITE-BACK$, \langle v', t' \rangle_{c'} \rangle)$
7: **wait** $receive(q, \langle$ACK$\rangle), \forall q \in Q'$
8: **return** v'
end function

Execução do servidor. A execução do servidor engloba os casos vistos para o servidor na seção 3.4.1.2 (algoritmo 7) e adiciona mais um caso quando do recebimento de uma mensagem WRITE-BACK de um cliente c. Neste caso, o servidor atualiza o seu estado somente se: *(i)* o par presente na mensagem WRITE-BACK estiver corretamente assinado por algum cliente c', que o escreveu em algum servidor correto $s' \in U$ antes da leitura e reescrita do mesmo par pelo cliente c; *(ii)* o *timestamp* $t_{s'}$ contido nesta mensagem for maior que o *timestamp* t_s já armazenado.

$\{c' \in \Pi$, tal que c' escreveu anteriormente $\langle v_{s'}, t_{s'} \rangle$ em um servidor correto $s' \in U\}$
upon $receive(c, \langle$WRITE-BACK$, \langle v_{s'}, t_{s'} \rangle_{c'} \rangle)$
1: **if** $sign\_ok(\langle v_{s'}, t_{s'} \rangle_{c'})$ **then**
2: **if** $t_{s'} > t_s$ **then**
3: $\langle v_s, t_s \rangle \leftarrow \langle v_{s'}, t_{s'} \rangle$
4: **end if**
5: **end if**
6: $send(c, \langle$ACK$\rangle)$

Complexidade de mensagens: os algoritmos MWMR atômico executam com complexidade de mensagens na ordem de $O(n)$. Ambas operações de escrita e leitura realizam-se em 4 passos de comunicação.

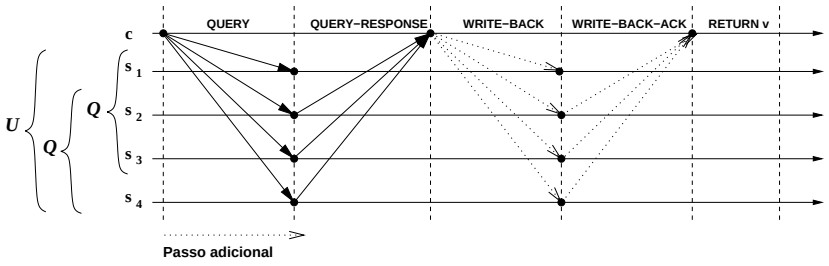


Figura 3.6: Protocolo de leitura – quóruns simétricos, clientes corretos e MWMR atômico para $f = 1$

3.4.2 Clientes faltosos

Esta seção apresenta protocolos que consideram a presença de clientes faltosos em um sistema de quóruns simétricos.

3.4.2.1 SWMR seguro [32]

Para clientes faltosos com semântica *single-writer multi-reader* segura, utiliza-se o sistema de quóruns de f-mascaramento (seção 3.2.3.1).

Funcionamento da escrita (algoritmo 9).

Do ponto de vista do cliente, a escrita é feita em apenas uma fase: cliente calcula o seu novo *timestamp t* maior que todos os seus *timestamps* já definidos (variável $last\_ts$). Em seguida, prepara a mensagem UPDATE contendo o par $\langle v,t\rangle$ e a envia para todos os servidores, esperando um conjunto de confirmações de um quórum Q.

Algoritmo 9 Escrita de um cliente c

procedure $write(v)$
1: $t \leftarrow min\{t_c \in T_c : last\_ts < t_c\}$
2: $last\_ts \leftarrow t$
3: $\forall s \in U$, $send(s, \langle \mathsf{UPDATE}, \langle v,t\rangle\rangle)$
4: **wait** $receive(q, \langle \mathsf{ACK}\rangle), \forall q \in Q$
end procedure

Funcionamento da leitura. O protocolo de leitura é igual ao algoritmo 4 visto na seção 3.4.1.1.

Execução do servidor (algoritmo 10). Diferentemente dos protocolos anteriores, a maior parte do protocolo de escrita é executada pelos servidores. Neste caso, após o cliente emitir sua mensagem de escrita, os servidores trocam um conjunto de mensagens ECHO e READY (ambas contendo o par $\langle v,t\rangle$ a ser escrito pelo cliente) com outros servidores a fim de manter consistência em seus valores armazenados.

Ao receber a mensagem QUERY de um cliente c, o servidor s devolve o seu par armazenado $\langle v_s, t_s\rangle$. Ao receber a mensagem UPDATE de um cliente c com o par $\langle v,t\rangle$ a ser escrito, o servidor s verifica se: *(i)* o *timestamp t* é um *timestamp* válido do cliente c ($t \in T_c$); *(ii)* ele não recebera um par $\langle v_{s'}, t_{s'}\rangle$ de c, onde $t_{s'} > t$ ou $(t_{s'} = t) \wedge (v_{s'} \neq v)$. Caso as condições *(i)* e *(ii)* sejam satisfeitas, o servidor s envia uma mensagem ECHO para todos os outros servidores.

Ao receber idênticas mensagens ECHO de um quórum de servidores, o servidor s envia uma mensagem READY para todos os servidores. E, por fim, o servidor s espera um conjunto de $|Q| - f$ mensagens idênticas READY de diferentes servidores corretos (pertencentes ao conjunto Q^-) contendo o par $\langle v,t\rangle$. Se o *timestamp t* for maior que o *timestamp* t_s armazenado em s, este servidor atualiza o seu estado com o par $\langle v,t\rangle$. Independentemente de esta condição ser satisfeita, o servidor envia ao cliente c uma mensagem de confirmação.

Algoritmo 10 Execução de um servidor s

upon $receive(c, \langle \mathsf{QUERY} \rangle)$
1: $send(c, \langle \mathsf{QUERY\text{-}RESPONSE}, \langle v_s, t_s \rangle \rangle)$

{*echoes* armazena todas as mensagens recebidas dos clientes e "ecoadas" por s até então}
upon $receive(c, \langle \mathsf{UPDATE}, \langle v, t \rangle \rangle)$
1: **if** $t \in T_c \wedge (\nexists \langle v_{s'}, t_{s'} \rangle \in echoes : (t_{s'} = t \wedge v_{s'} \neq v) \vee (t_{s'} > t))$ **then**
2: $\quad \forall s \in U$, $send(s, \langle \mathsf{ECHO}, \langle v, t \rangle \rangle)$
3: **end if**

upon $\forall s \in Q$, $receive(s, \langle \mathsf{ECHO}, \langle v, t \rangle \rangle)$
1: $\forall s \in U$, $send(s, \langle \mathsf{READY}, \langle v, t \rangle \rangle)$

upon $\forall s \in Q^{-}$, $receive(s, \langle \mathsf{READY}, \langle v, t \rangle \rangle)$ $\{|Q^{-}| = |Q| - f\}$
1: **if** $t > t_s$ **then**
2: $\quad \langle v_s, t_s \rangle \leftarrow \langle v, t \rangle$
3: **end if**
4: $send(c, \langle \mathsf{ACK} \rangle)$

Complexidade de mensagens: a escrita (algoritmo 9) tem complexidade de mensagem $O(n^2)$. A leitura, assim como o algoritmo 4 da seção 3.4.1.1, tem complexidade de mensagem $O(n)$. As operações de escrita e leitura realizam-se, respectivamente, em 4 e 2 passos de comunicação.

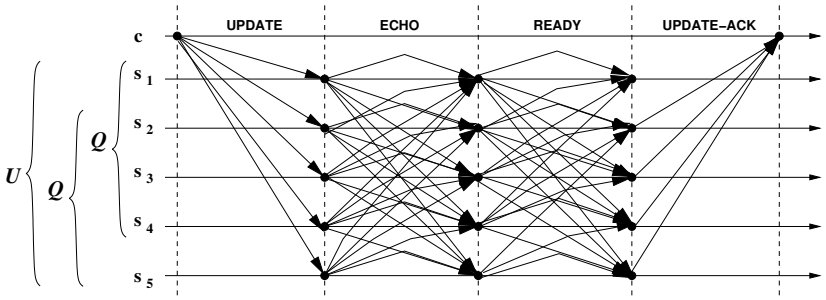


Figura 3.7: Protocolo de escrita – quóruns simétricos, clientes faltosos e SWMR seguro para $f = 1$

3.4.2.2 MWMR seguro [29]

Assim como na seção 3.4.2.1, usa-se o sistema de quóruns de f-mascaramento (seção 3.2.3.1), mas com semântica *multi-writer multi-reader* segura. Neste caso, cada servidor utiliza um par de chaves privada (para assinatura) e pública (para verificação).

Funcionamento da escrita (algoritmo 11). O protocolo de escrita se completa em 3 fases. Na segunda fase do protocolo, ao receberem uma mensagem de escrita do cliente com o par $\langle v, t \rangle$ a ser escrito, os servidores devolvem este mesmo par $\langle v, t \rangle$ assinado para receberem, na terceira fase, deste mesmo cliente, uma mensagem contendo novamente aquele mesmo par e uma lista com os ecos assinados por um quórum de servidores. Através deste mecanismo, clientes faltosos não conseguem escrever diferentes valores em servidores corretos. Este mecanismo de validação de consistência de dados é também conhecido como *Echo Broadcast* [40].

Fase 1: cliente requisita um conjunto de pares ao quórum Q chamando o procedimento *query* (algoritmo 1) e armazenando o resultado em um vetor S inicialmente vazio. Em seguida, o cliente

define o menor valor do *timestamp* $t \in T_c$ que seja maior do que todos os seus *timestamps* já definidos (variável *last_ts*) e maior do que todos os *timestamps* recebidos do quórum Q;

Fase 2: cliente prepara o par $\langle v,t \rangle$, envia a mensagem UPDATE contendo este par para todos os servidores (linha 5) e espera um conjunto E de ecos assinados de $\langle v,t \rangle$ de todos os servidores de um quórum Q' (linhas 6 a 10);

Fase 3: cliente envia mensagem READY para todos os servidores contendo o conjunto E com pares $\langle v,t \rangle$ assinados pelo quórum Q' na fase 2 e propriamente o par $\langle v,t \rangle$ a ser escrito pelo cliente (linha 11). Por fim, o cliente espera um conjunto de confirmações de um quórum de servidores.

Algoritmo 11 Escrita de um cliente c

procedure *write*(v)
1: $S \leftarrow query(|Q|)$
2: $max\_ts \leftarrow max\{S[\,].ts\}$
3: $t \leftarrow min\{t_c \in T_c : max\_ts < t \wedge last\_ts < t\}$
4: $last\_ts \leftarrow t$
5: $\forall s \in U$, $send(s, \langle \mathsf{UPDATE}, \langle v,t \rangle \rangle)$
6: $E[1 \ldots n] \leftarrow \perp$
7: **repeat**
8: **wait** $receive(s, \langle \mathsf{ECHO}, \langle v,t \rangle_s \rangle)$, $\forall s \in Q'$
9: $E[s] \leftarrow \{\langle \mathsf{ECHO}, \langle v,t \rangle_s \rangle\}$
10: **until** $\#_{\perp} E \leq n - |Q|$
11: $\forall s \in U$, $send(s, \langle \mathsf{READY}, E, \langle v,t \rangle \rangle)$
12: **wait** $receive(s, \langle \mathsf{ACK} \rangle)$, $\forall s \in Q''$
end procedure

Funcionamento da leitura (algoritmo 12). A leitura de um cliente é efetuada em 2 fases e é similar ao algoritmo 4 da seção 3.4.1.1. A diferença aqui estão no recebimento do cliente de pares armazenados assinados pelos servidores. Neste caso, os clientes não verificam a autenticidade dos pares recebidos, uma vez que não mantêm a chave pública dos servidores. No algoritmo 4, diferentemente do algoritmo 12, os servidores não respondiam a consulta com valores assinados e não havia mensagem de reescrita (*write back*).

No caso da reescrita, o cliente envia o par a ser reescrito e uma lista L com, pelo menos, $f+1$ ocorrências deste par assinadas por servidores $s' \in Q'$ que estão no quórum de leitura Q ($Q' \subseteq Q$). Esta lista serve como justificativa da reescrita e evita que leitores maliciosos escrevam, por exemplo, valores diferentes em cada servidor correto no quórum durante a ação de reescrita. No retorno do *write back*, o cliente espera um conjunto de confirmações de um quórum.

Fase 1: similar à fase 1 do algoritmo 4. **Fase 2**: cliente envia mensagem WRITE-BACK para todos os servidores contendo o par $\langle v,t \rangle$ e uma lista L com as suas $f+1$ ocorrências coletadas na fase 1 do protocolo (linha 5). Em seguida, o cliente espera por um conjunto de confirmações de um quórum de servidores antes de retornar o valor lido (linhas 5 e 6). Caso o cliente não encontre $f+1$ ocorrências iguais de um par $\langle v,t \rangle$ (em caso de leitura concorrente com escrita), a operação devolve $\perp$ indicando falha na leitura (linha 8).

Execução do servidor (algoritmo 13). Quando o servidor recebe QUERY, ele devolve assinando o seu par armazenado $\langle v,t \rangle$. Quando o servidor recebe UPDATE de um cliente c contendo um par

Algoritmo 12 Leitura de um cliente c

value **function** *read*()
1: $S \leftarrow query\_w\_sign(|Q|)$
2: **if** $\#_{\langle v,t\rangle} S \geq f+1$ **then**
3: $\quad L \leftarrow S_{\langle v,t\rangle}$
4: $\quad \forall s \in U$, *send*(s,WRITE-BACK,$L, \langle v,t\rangle\rangle$)
5: $\quad$ **wait** *receive*($q, \langle$ACK$\rangle$), $\forall q \in Q'$
6: $\quad$ **return** v
7: **else**
8: $\quad$ **return** $\perp$
9: **end if**
end function

$\langle v,t\rangle$, aquele verifica se o *timestamp* $t \in T_c$ e se existe um par $\langle v_e,t_e\rangle$ já enviado por c no conjunto *echoed* (conjunto de pares já "ecoados" pelo servidor) com o mesmo *timestamp* ($t_e = t$) e com valores diferentes ($v_e \neq v$). Se tal par $\langle v_e,t_e\rangle$ não existir, s devolve um eco assinado de $\langle v,t\rangle$ (linha 4).

Quando um servidor s recebe uma mensagem READY contendo um conjunto E de ecos assinados e um par $\langle v,t\rangle$ a ser escrito por um cliente c, as seguintes condições são verificadas: *(i)* se há um quórum de ecos assinados em E; *(ii)* se as assinaturas dos ecos em E são autênticas; *(iii)* se um número de $f+1$ servidores ecoaram o mesmo par $\langle v,t\rangle$ enviado pelo cliente c em READY. Se as condições *(i)*, *(ii)* e *(iii)* forem satisfeitas, e se o *timestamp* t for maior que o t_s armazenado em s, o servidor atualiza o seu estado (linha 5);

Na leitura, quando um servidor s recebe uma mensagem WRITE-BACK de um cliente c contendo o par $\langle v,t\rangle$ consultado do quórum mais uma lista L com suas $f+1$ assinaturas, as seguintes condições são verificadas: se *(i)* existem $f+1$ pares assinados em L; *(ii)* se as $f+1$ assinaturas em L são autênticas; *(iii)* se os pares assinados correspondem ao par $\langle v,t\rangle$ sendo reescrito por c. Caso as condições *(i)*, *(ii)* e *(iii)* sejam satisfeitas, o servidor atualiza o seu estado somente se o *timestamp* t for maior que o seu *timestamp* t_s armazenado.

Complexidade de mensagens: a escrita e a leitura (algoritmo 11 e algoritmo 12, respectivamente) têm complexidade de mensagem $O(n)$. A atualização do servidor (algoritmo 13) tem complexidade de mensagem de $O(n)$. As operações de escrita e leitura realizam-se, respectivamente, em 6 e 4 passos de comunicação.

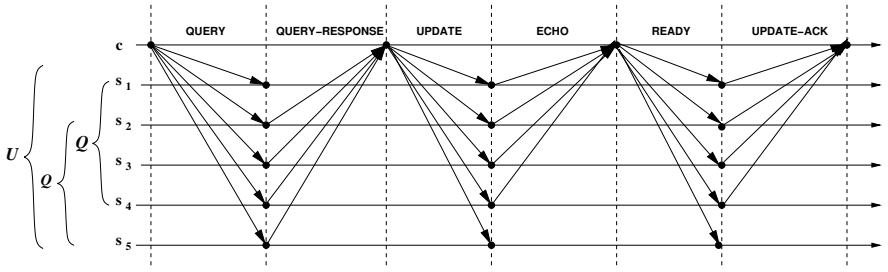


Figura 3.8: Protocolo de escrita – quóruns simétricos, clientes faltosos e MWMR seguro para $f = 1$

Algoritmo 13 Execução de um servidor s

upon *receive*$(c, \langle \mathsf{QUERY} \rangle)$
1: *send*$(c, \langle \mathsf{QUERY\text{-}RESPONSE} \langle v_s, t_s \rangle, proof \rangle)$

upon *receive*$(c, \langle \mathsf{UPDATE}, \langle v,t \rangle \rangle)$
1: $\{$seja $\langle v_e, t_e \rangle$ um par tal que $(t_e = t) \wedge (v_e \neq v)\}$
2: **if** $t \in T_c \wedge \langle v_e, t_e \rangle \notin$ *echoed* **then**
3: $\quad$ *echoed* $\leftarrow$ *echoed* $\cup \{\langle v,t \rangle\}$
4: $\quad$ *send*$(c, \langle \mathsf{ECHO}, \langle v,t \rangle_s \rangle)$
5: **end if**

upon *receive*$(c, \langle \mathsf{READY}, E, \langle v,t \rangle \rangle)$
1: **if** $[\exists E' \subseteq E : |E'| \geq f+1 \wedge valid(\langle v,t \rangle, E')]$ **then**
2: $\quad$ **if** $t > t_s$ **then**
3: $\quad\quad \langle v_s, t_s \rangle \leftarrow \langle v,t \rangle$
4: $\quad$ **end if**
5: $\quad$ *send*$(c, \langle \mathsf{ACK} \rangle)$
6: **end if**

upon *receive*$(c, \langle \mathsf{WRITE\text{-}BACK}, L, \langle v,t \rangle \rangle)$
1: **if** $[\exists L' \subseteq L_{\langle v,t \rangle} : |L'| \geq f+1 \wedge valid(\langle v,t \rangle, L')]$ **then**
2: $\quad$ **if** $t > t_s$ **then**
3: $\quad\quad \langle v_s, t_s \rangle \leftarrow \langle v,t \rangle$
4: $\quad$ **end if**
5: $\quad$ *send*$(c, \langle \mathsf{ACK} \rangle)$
6: **end if**

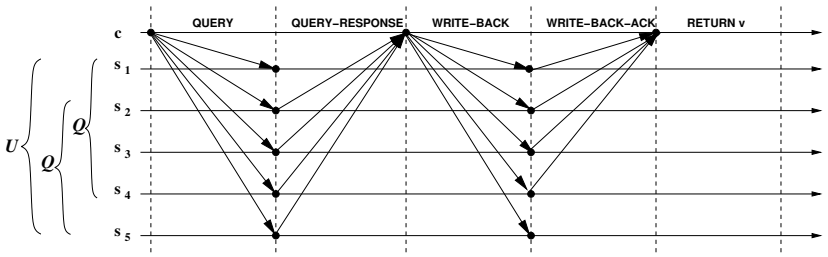


Figura 3.9: Protocolo de leitura – quóruns simétricos, clientes faltosos e MWMR seguro para $f = 1$

3.4.2.3 MWMR atômico [27]

Os algoritmos neste caso atuam em sistemas de quóruns de f-disseminação (seção 3.2.3.2). A semântica de consistência neste caso é *multi-writer multi-reader* atômica.

Para este algoritmo, clientes e servidores mantêm algumas variáveis locais específicas. No servidor, existem as seguintes variáveis:

- P: lista que contém o último par $\langle v,t \rangle$ que foi preparado para escrita por cada cliente no servidor. Notação (para um cliente c): $P[c].t$ (*timestamp* preparado) e $P[c].v$ (valor preparado);
- O: lista similar a P. Utilizado apenas no protocolo otimizado de escrita. Contém os últimos pares $\langle v,t \rangle$ preparados pelo servidor em nome de cada cliente;
- *write-ts*: *timestamp* da última escrita consolidada neste servidor.

O cliente mantém apenas uma variável:

- W: vetor com $2f+1$ mensagens UPDATE-ACK assinadas contendo um mesmo *timestamp* t referente à última operação de escrita concluída e consolidada pelo cliente no sistema. Notação: uma mensagem UPDATE-ACK vinda de um servidor s é referida por $W[s]$. O *timestamp* contido na mensagem emitida pelo servidor s é identificado por $W[s].t$. O *timestamp* representando todos os *timestamps* presentes em W é identificado por $W.t$.

Funcionamento da escrita (protocolo normal). A escrita normal possui três fases (veja procedimento *write_normal* no algoritmo 14). Para executar cada passo da escrita, o cliente precisa apresentar um conjunto de $2f+1$ mensagens assinadas, coletadas de um quórum, comprovando que é capaz de realizar aquele passo e justificando as suas próximas ações. Por exemplo, o cliente tem que atestar que concluiu a sua última escrita antes de realizar uma nova escrita, ou, então, atestar que usa *timestamps* válidos para calcular um novo *timestamp* antes de prosseguir na sua nova escrita.

Fase 1: o cliente requisita um conjunto de pares auto-verificáveis de um quórum Q, que são armazenados no vetor S (inicialmente vazio) usando uma função *query_w_sign* (algoritmo 2). Cliente seleciona o maior *timestamp* contido em um par válido e calcula o seu novo $timestamp$ maior do que todos os seus *timestamps* já recebidos (variável *last_ts*) e maior do que todos os *timestamps* válidos recebidos de Q (linhas 1 a 4);

Fase 2: cliente prepara um novo par $\langle v,t\rangle$ e o envia numa mensagem PREPARE para todos os servidores juntamente com o certificado do *timestamp* utilizado para gerar t (certificado *proof* obtido na fase 1, ver detalhes no algoritmo 2), além da prova W da sua última escrita (ou nulo se o cliente não realizou escrita alguma). Cliente espera mensagens PREPARE-ACK válidas (corretamente assinadas e com os pares valor-*timestamp* correspondentes a $\langle v,t\rangle$) de um quórum de servidores e constrói com estas mensagens a sua prova de preparação S_{proofs} (linhas 5 a 7).

Fase 3: cliente envia uma mensagem UPDATE com a prova de preparação obtida na fase 2 e o novo valor. Cliente espera mensagens UPDATE-ACK válidas (corretamente assinadas e com o mesmo *timestamp*) de um quórum Q'. Estas mensagens são armazenadas na sua variável local W (linhas 8 a 10).

Algoritmo 14 Escrita de um cliente c (protocolo normal)

procedure *write_normal*(v)
1: $S \leftarrow query\_w\_sign(|Q|)$
2: $max\_ts \leftarrow max\{S[\,].ts\}$
3: $t \leftarrow min\{t_c \in T_c : max\_ts < t_c \wedge last\_ts < t_c\}$
4: $last\_ts \leftarrow t$
5: $\forall s \in U$, $send(s, \langle \text{PREPARE}, \langle v,t\rangle, proof, W\rangle)$
6: **wait** $receive(q, \langle \text{PREPARE-ACK}, \langle v,t\rangle_q\rangle), \forall q \in Q$
7: $S_{proofs} \leftarrow \{\forall q \in Q, \langle \text{PREPARE-ACK}, \langle v,t\rangle_q : valid(\langle \text{PREPARE-ACK}, \langle v,t\rangle_q, \langle v,t\rangle)\}$
8: $\forall s \in U$, $send(s, \langle \text{UPDATE}, v, S_{proofs}\rangle)$
9: **wait** $receive(q, \langle \text{UPDATE-ACK}, t\rangle_q), \forall q \in Q'$
10: $W \leftarrow \{\forall q \in Q', \langle \text{UPDATE-ACK}, t\rangle_q : valid(\langle \text{UPDATE-ACK}, t\rangle_q, t)\}$
end procedure

Funcionamento da escrita (protocolo otimizado). A escrita otimizada (procedimento *write_opt* no algoritmo 15) é realizada em 2 fases em princípio. Primeiro, o cliente tenta efetuar as fases 1 e 2 como uma única fase. Caso não consiga, executa o protocolo normal de escrita (3 fases). No caso de execução otimizada, o *timestamp* é calculado nos servidores em nome do cliente.

Fase 1: cliente envia mensagem READ-TS-PREP para todos os servidores com o valor proposto v e sua prova de escrita. Cliente espera receber um conjunto de mensagens READ-PREP-ACK válidas (corretamente assinadas) de um quórum de servidores contendo o par $\langle v,t \rangle$. Estas mensagens assinadas são armazenadas em um conjunto S inicialmente vazio (linhas 1 a 4);

Fase 2: se o cliente receber pares assinados de um quórum de servidores com o mesmo *timestamp* (conjunto S'), executa imediatamente a fase 3 do protocolo normal (passo de escrita) usando S' como a prova da preparação (S_{proofs}). Caso contrário, escolhe o maior *timestamp* entre as mensagens READ-PREP-ACK e realiza a fase 2 do protocolo normal (passo de preparação, linhas 8 a 13). **Fase 3**: igual à fase 3 do protocolo normal.

Algoritmo 15 Escrita de um cliente c (protocolo otimizado)

procedure *write_opt*(v)
1: $S = \emptyset$
2: $\forall s \in U$, $send(s, \langle \text{READ-TS-PREP}, v, W \rangle)$
3: **wait** $receive(q, \langle \text{READ-PREP-ACK}, \langle v_q, t_q \rangle \rangle_q), \forall q \in Q$
4: $S \leftarrow \{\forall q \in Q, \langle \text{READ-PREP-ACK}, \langle v_q, t_q \rangle \rangle_q : sign\_ok(\langle \text{READ-PREP-ACK}, \langle v_q, t_q \rangle \rangle_q)\}$
5: $t \leftarrow \{\forall q \in Q, \exists t' : \langle \text{READ-PREP-ACK}, \langle v_q, t' \rangle \rangle_q\}$
6: $S' \leftarrow [S_{\langle \text{READ-PREP-ACK}, \langle v_q, t \rangle \rangle_q}], \forall q \in Q$
7: **if** $|S'| < |Q|$ **then**
8: $\quad max\_ts \leftarrow \{\forall \langle \text{READ-PREP-ACK}, \langle v_s, t_s \rangle \rangle_s \in S, max\{t_s\}\}$
9: $\quad t \leftarrow min\{t_c \in T_c : max\_ts < t_c \wedge last\_ts < t_c\}$
10: $\quad last\_ts \leftarrow t$
11: $\quad \forall s \in U$, $send(s, \langle \text{PREPARE}, \langle v, t \rangle, proof, W \rangle)$
12: $\quad$ **wait** $receive(q, \langle \text{PREPARE-ACK}, \langle v_q, t_q \rangle \rangle_q), \forall q \in Q$
13: $\quad S_{proofs} \leftarrow \{\forall q \in Q, \langle \text{PREPARE-ACK}, \langle v_q, t_q \rangle \rangle_q : sign\_ok(\langle \text{PREPARE-ACK}, \langle v_q, t_q \rangle \rangle_q)\}$
14: **else**
15: $\quad S_{proofs} \leftarrow S'$
16: **end if**
17: $\forall s \in U$, $send(s, \langle \text{UPDATE}, v, S_{proofs} \rangle)$
18: **wait** $receive(q, \langle \text{UPDATE-ACK}, t \rangle_q), \forall q \in Q'$
19: $W \leftarrow \{\forall q \in Q', \langle \text{UPDATE-ACK}, t \rangle_q : sign\_ok(\langle \text{UPDATE-ACK}, t \rangle_q)\ \}$
end procedure

Funcionamento da leitura (algoritmo 16). A leitura do cliente (função *read*) executa em uma ou duas fases, a depender de o cliente realizar ou não uma reescrita (*write back*) em um quórum de servidores.

Fase 1: cliente requisita um conjunto de pares assinados a um quórum Q usando uma função *query_w_sign* (algoritmo 2). Cliente espera um quórum de respostas válidas, que são armazenadas no conjunto S (inicialmente vazio) e seleciona aquela com o maior *timestamp*.

Fase 2: se os *timestamps* retornados da fase 1 forem diferentes, o cliente envia uma mensagem WRITE-BACK assinada contendo um certificado *proof* do *timestamp max_ts* (maior *timestamp* encontrado nos pares do conjunto S obtido na fase 1) e do valor v' (maior valor entre os pares do conjunto

$S' \subseteq S$ com *timestamps* iguais a *max_ts*). Esta mensagem de WRITE-BACK é enviada apenas para os servidores que não responderam ao cliente na fase 1 com um par válido $\langle v', max\_ts \rangle$. O valor v' é obtido pelo fato de possivelmente existirem valores diferentes para o mesmo *timestamp* caso o protocolo otimizado de escrita seja usado. No final, o cliente espera mensagens válidas (corretamente assinadas e com o mesmo *timestamp*) de um quórum. O conjunto destas mensagens recebidas formam a prova de escrita do cliente (variável W). A mensagem de reescrita é o que garante a semântica atômica do protocolo, uma vez que todos os clientes lerão o último valor escrito (o da reescrita) até ocorrer a próxima escrita.

Algoritmo 16 Leitura de um cliente c

value **function** *read*()
1: $S \leftarrow query\_w\_sign(|Q|)$
2: $max\_ts \leftarrow max\{S[\,].ts\}$
3: $S' \leftarrow \{\forall q \in Q, S_{\langle v_q, max\_ts \rangle}\}$
4: $v' \leftarrow max\{S'[\,].v\}$
5: $S'' \leftarrow \{\forall s' : S[s'] = \langle v', max\_ts \rangle\}$
6: **if** $|S''| < |Q|$ **then**
7: $\quad \forall s \in U \setminus S'', send(s, \langle \text{WRITE-BACK}, v', proof \rangle)$
8: $\quad$ **wait** $receive(q, \langle \text{UPDATE-ACK}, t \rangle_q), \forall q \in Q'$
9: $\quad W \leftarrow \{\forall q \in Q', \langle \text{UPDATE-ACK}, t \rangle_q : sign\_ok(\langle \text{UPDATE-ACK}, t \rangle_q)\}$
10: **end if**
11: **return** v'
end function

Execução do servidor, parte 1 (algoritmo 17). Quando um servidor recebe uma mensagem QUERY de um cliente, aquele devolve o seu par armazenado mais o certificado deste par. Quando um servidor s recebe uma mensagem válida READ-TS-PREP de um cliente c com um valor v_c (protocolo otimizado de escrita), s verifica se a prova de escrita do cliente (W) contida na mensagem é válida: um quórum de mensagens UPDATE-ACK corretamente assinadas e com o mesmo *timestamp* (linha 1). Em caso positivo, s atualiza a sua variável *write-ts* se o *timestamp* referente a W ($W.t$) for maior do que *write-ts* (linha 2). Depois, o servidor calcula um novo *timestamp next_ts* $\in T_c$, maior do que o seu *timestamp* armazenado t_s, e tenta preparar a escrita otimizada do cliente c (função *update_list* no algoritmo 19).

Em princípio, o servidor atualiza as listas O e P mantendo apenas os registros com *timestamps* maiores do que *write-ts* (linhas 1 a 5 do algoritmo 19). Para que a preparação ocorra, não pode existir um registro de c nas listas O e P do servidor s, a menos que seja um registro com *timestamp* e valor iguais a *next_ts* e v_c, respectivamente (para que c não tenha duas escritas diferentes em andamento em um mesmo servidor). No caso de não existir um registro de c em O, se a escrita em preparação (representada pelo *timestamp next_ts*) for mais recente do que a última escrita feita no sistema (representada pelo *timestamp write-ts*), o servidor adiciona um registro com o par $\langle v_c, next\_ts \rangle$ somente na lista O (linha 12 do algoritmo 19). Se a preparação ocorrer com sucesso, o servidor responde com uma mensagem READ-PREP-ACK assinada contendo o par preparado para escrita (linha 8). Caso contrário (preparação falhou), o servidor retorna seu par armazenado com o certificado correspondente, tal como em uma resposta à mensagem QUERY (linha 11).

Execução do servidor, parte 2 (algoritmo 18). Quando um servidor s recebe uma mensagem

algoritmo 17 Execução de um servidor s (parte 1)

$\{is\_opt\_protocol$: indica se o protocolo otimizado é usado$\}$
upon $receive(c, \langle \text{QUERY} \rangle)$
1: $send(c, \langle \text{QUERY-RESPONSE}, \langle v_s, t_s \rangle, proof \rangle)$
upon $receive(c, \langle \text{READ-TS-PREP}, v_c, W \rangle)$
1: **if** $[\forall s' \in Q, sign\_ok(W[s'])] \wedge \#_t W = 2f + 1$ **then**
2: $write\text{-}ts \leftarrow max\{write\text{-}ts, W.t\}$
3: $next\_ts \leftarrow min\{t \in T_c : t > t_s\}$
4: $o\_status \leftarrow update\_list(O, c, \langle v_c, next\_ts \rangle, true)$
5: $p\_status \leftarrow update\_list(P, c, \langle v_c, next\_ts \rangle, false)$
6: **if** $o\_status \wedge p\_status$ **then**
7: $is\_opt\_protocol \leftarrow true$
8: $send(c, \langle \text{READ-PREP-ACK}, \langle v_c, next\_ts \rangle \rangle_s)$
9: **else**
10: $is\_opt\_protocol \leftarrow false$
11: $send(c, \langle \text{READ-PREP-ACK}, \langle v_s, t_s \rangle \rangle_s)$
12: **end if**
13: **end if**

válida PREPARE de um cliente c, que prepara a escrita de um par $\langle v_c, t_c \rangle$, aquele executa passos similares ao caso visto no algoritmo 17, em que s recebe uma mensagem READ-TS-PREP. Neste caso, porém, o servidor não cria o *timestamp* em nome do cliente, uma vez que o cliente já o fizera e o enviou dentro da mensagem PREPARE.

Sendo assim, se a prova de escrita do cliente (W) e o certificado *proof* forem válidos, o servidor verifica se o *timestamp* enviado por c é válido: se $t_c \in T_c$ e se t_c é o sucessor de um *timestamp* $t_p \in proof$ (linhas 2 e 3). Tal verificação serve para impedir que clientes maliciosos tentem escrever dados com *timestamps* definidos arbitrariamente (em alguns casos, atribuindo valores extremamente grandes, o que impossibilitaria a realização de novas escritas no sistema). Se t_c for válido, o servidor atualiza a variável *write-ts* (linha 4) tal como no caso em que o servidor recebe uma mensagem READ-TS-PREP (algoritmo 17).

Aqui, o servidor apenas atualiza a sua lista P mantendo os registros com *timestamps* maiores do que *write-ts*. Os critérios de preparação de escrita seguem o descrito na parte 1 da execução do servidor. A preparação da escrita do par $\langle v_c, t_c \rangle$ pelo cliente c só ocorre se não existir registro de escrita preparada para c em P. Nesta situação, o servidor atualiza $P[c]$ com $\langle v_c, t_c \rangle$ se esta escrita em preparação for mais recente que a última escrita realizada em um quórum (linha 11 do algoritmo 19). Caso contrário (existe um registro de c em P), a preparação só é efetivada se este registro for para um par cujo valor e *timestamp* sejam iguais a v_c e t_c, respectivamente, o que impede um cliente de prosseguir com uma nova escrita sem ter terminado uma anterior. Nos casos de a preparação ocorrer com sucesso, o servidor retorna uma mensagem PREPARE-ACK assinada com o par $\langle v_c, t_c \rangle$.

Outro caso de execução de um servidor s é quando este recebe uma mensagem válida UPDATE de um cliente c, que deseja escrever um valor já preparado v_c. Inicialmente, o servidor verifica se a prova de preparação do cliente (S_{proofs}) é válida: se há um quórum de mensagens PREPARE-ACK corretamente assinadas, com o mesmo par valor-*timestamp* ($S_{proofs}.v$ e $S_{proofs}.t$, respectivamente) e se todos os valores são iguais a v_c (linhas 1 e 2). Se S_{proofs} for válido, o servidor tenta atualizar o seu estado. No protocolo normal, a atualização acontece se o *timestamp* correspondente à prova

de preparação ($S_{proofs}.t$) for maior do que o *timestamp* t_s armazenado no servidor. No protocolo otimizado, o servidor atualiza-se efetivamente caso o *timestamp* da prova de preparação for igual (é possível que algum cliente já tenha escrito algum valor com o mesmo *timestamp* pelo protocolo normal) ou maior ao seu *timestamp* t_s armazenado em s. Por fim, s devolve uma mensagem UPDATE-ACK assinada contendo o *timestamp* referente à prova de preparação do cliente, mesmo que a escrita não se realize de fato.

Caso receba uma mensagem WRITE-BACK, o servidor executa como em uma mensagem UPDATE.

algoritmo 18 Execução de um servidor s (parte 2)

$\{is\_opt\_protocol$: indica se o protocolo otimizado é usado$\}$
upon $receive(c, \langle \text{PREPARE}, \langle v_c, t_c \rangle, proof, W \rangle)$
1: **if** $[\forall s' \in Q, sign\_ok(W[s'])] \wedge \#_t W = 2f+1$ **then**
2: $[prev \leftarrow max\{t' \in T_c : t' < t_c]$
3: **if** $(prev = t_p : t_p \in proof)$ **then**
4: $write\text{-}ts \leftarrow max\{write\text{-}ts, W.t\}$
5: $p\_status \leftarrow update\_list(P, c, \langle v_c, t_c \rangle, true)$
6: **if** p_status **then**
7: $is\_opt\_protocol \leftarrow false$
8: $send(c, \langle \text{PREPARE-ACK}, \langle v_c, t_c \rangle \rangle_s)$
9: **end if**
10: **end if**
11: **end if**
upon $receive(c, \langle \text{UPDATE}, v_c, S_{proofs} \rangle)$
1: **if** $[\forall s' \in Q, sign\_ok(S_{proofs}[s'])] \wedge \#_{\langle v,t \rangle} S_{proofs} = 2f+1$ **then**
2: **if** $(\forall s' \in Q, v_c = S_{proofs}[s'].v)$ **then**
3: **if** $S_{proofs}.t > t_s$ **then**
4: $\langle v_s, t_s \rangle \leftarrow \langle v, S_{proofs}.t \rangle$
5: **else if** $S_{proofs}.t = t_s$ **then**
6: **if** $is\_opt\_protocol$ **then**
7: $\langle v_s, t_s \rangle \leftarrow \langle max\{v_s, v\}, t_s \rangle$
8: **end if**
9: **end if**
10: $send(c, \langle \text{UPDATE-ACK}, S_{proofs}.t \rangle_s)$
11: **end if**
12: **end if**

Complexidade de mensagens: todos os algoritmos (leitura e escrita do cliente e algoritmo do servidor) ocorrem com complexidade de mensagens na ordem de $O(n)$. As operações de escrita e leitura, neste caso, se completam, respectivamente, em 6 e 4 passos de comunicação, para o protocolo normal; ou em 4 e 2 passos de comunicação, respectivamente, para o protocolo otimizado.

algoritmo 19 Atualização das listas P (escrita normal) e O (escrita otimizada) num servidor s

boolean **function** $update\_list(L, c, \langle v_c, t_c \rangle, update)$

```
1:  for all cliente c_i do
2:     if L[c_i].t ≤ write-ts then
3:        L[c_i] ← ⊥
4:     end if
5:  end for
6:  if ∃L[c] then
7:     if (v_c = L[c].v) ∧ (t_c = L[c].t) then
8:        return true
9:     end if
10: else
11:    if update ∧ t_c > write-ts then
12:       L[c] ← ⟨v_c, t_c⟩
13:    end if
14:    return true
15: end if
16: return false
end function
```

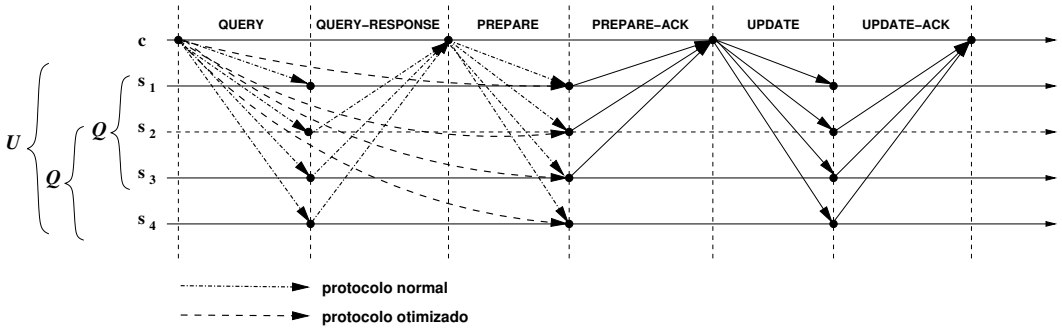


Figura 3.10: Protocolo de escrita – quóruns simétricos, clientes faltosos e MWMR atômico para $f = 1$

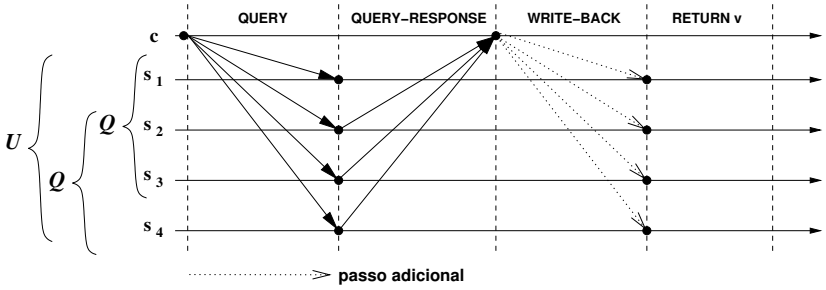


Figura 3.11: Protocolo de leitura – quóruns simétricos, clientes faltosos e MWMR atômico para $f = 1$

3.5 Algoritmos para sistemas de quóruns assimétricos

Esta seção descreve os algoritmos de armazenamento em sistema de quóruns bizantinos assimétricos, ou seja, com quóruns de leitura e escrita de tamanho diferentes.

Os algoritmos de leitura e escrita para os sistemas de quóruns assimétricos apresentados nesta seção assemelham-se aos usados nos seus respectivos casos simétricos. Um ponto fundamental que

diferencia o caso assimétrico do simétrico é a ausência de mensagens de confirmação nos passos de escrita. No modelo do sistema, já se consideram canais confiáveis, portanto a espera por confirmação da operação de escrita seria a princípio um passo descartável. A falta de mensagens de confirmação nas operações de escrita leva a uma diminuição de f servidores no sistema, refletindo na relação entre os quóruns de leitura e escrita, que possuem tamanhos diferentes.

3.5.1 Clientes corretos

Esta seção apresenta os algoritmos de BQS em quóruns assimétricos que não toleram clientes faltosos.

3.5.1.1 MWMR seguro [35]

Os procedimentos de escrita e leitura são realizados em sistema de quóruns de a-mascaramento (seção 3.2.3.3). A semântica de consistência é *multi-writer multi-reader* segura.

Funcionamento da escrita. O protocolo de escrita segue similar ao funcionamento do algoritmo 3 na seção 3.4.1.1. A Figura 3.12 ilustra o seu funcionamento.

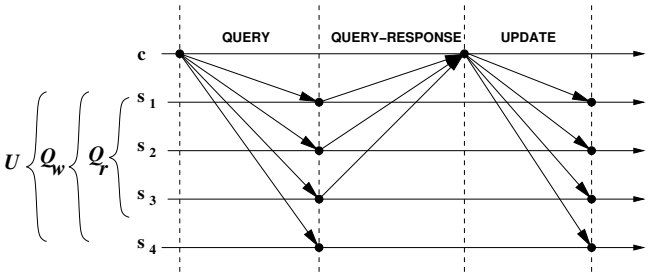


Figura 3.12: Protocolo de escrita – quóruns assimétricos, clientes corretos e MWMR seguro para $f = 1$

Funcionamento da leitura. O protocolo de leitura é também similar ao protocolo de leitura mostrado no algoritmo 4 (seção 3.4.1.1). A Figura 3.13 ilustra o seu funcionamento em quórum assimétrico.

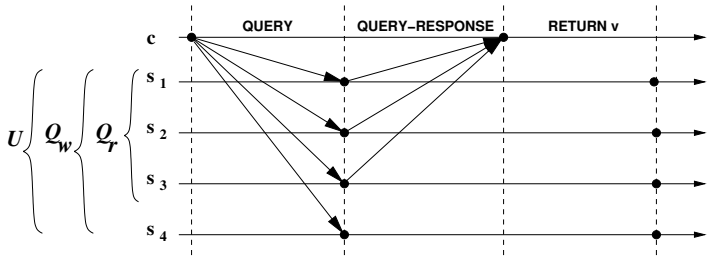


Figura 3.13: Protocolo de leitura – quóruns assimétricos, clientes corretos e MWMR seguro para $f = 1$

Execução do servidor. A atualização no servidor *s* funciona como o algoritmo 5 (seção 3.4.1.1), mas sem o caso com confirmação de mensagens.

Complexidade de mensagens: os algoritmos de escrita e leitura do cliente e o algoritmo do servidor possuem complexidade de troca de mensagens $O(n)$. As operações de escrita e leitura são realizadas, respectivamente, em 3 e 2 passos de comunicação.

3.5.1.2 MWMR regular [35]

Neste caso, a escrita e a leitura acontecem em sistema de quóruns de a-disseminação (seção 3.2.3.4). A semântica de consistência alcançada é *multi-writer multi-reader* regular uma vez que se utilizam dados auto-verificáveis.

Funcionamento da escrita. O procedimento de escrita funciona tal como o algoritmo de escrita na seção 3.4.1.2 (algoritmo 3), mas, como não existe confirmação da escrita (envio de mensagens UPDATE-ACK por parte dos servidores), o cliente não espera após enviar sua mensagem UPDATE.

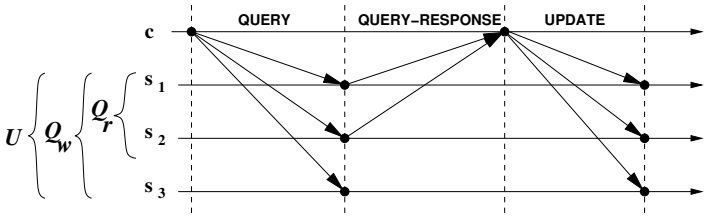


Figura 3.14: Protocolo de escrita – quóruns assimétricos, clientes corretos e MWMR regular para $f = 1$

Funcionamento da leitura. O algoritmo de leitura é idêntico ao algoritmo 6 apresentado na seção 3.4.1.2.

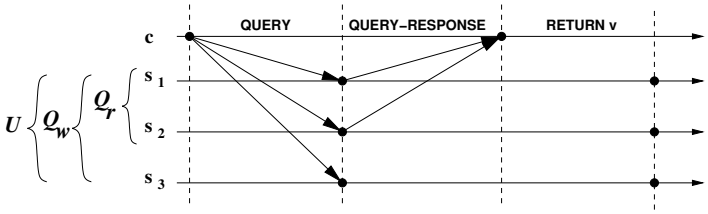


Figura 3.15: Protocolo de leitura – quóruns assimétricos, clientes corretos e MWMR regular para $f = 1$

Execução do servidor. A atualização de um servidor *s* neste caso segue o funcionamento do algoritmo 5, mas sem confirmação de mensagens.

Complexidade de mensagens: os algoritmos de escrita e leitura, bem como o algoritmo do servidor, possuem complexidade de troca de mensagens $O(n)$. A escrita e a leitura, respectivamente, ocorre em 3 e 2 passos de comunicação.

3.6 Sistema com quóruns “mínimos”

Esta seção descreve os algoritmos SBQ-L para sistema de quóruns mínimos, isto é, sistemas com quóruns assimétricos que utilizam o número mínimo necessário de servidores para se obter, pelo menos, fracas semânticas de consistência [34]: $3f+1$ servidores no caso dos protocolos com escritas confirmáveis e $2f+1$ servidores para protocolos com escritas não confirmáveis.

A depender do caso apresentado, os clientes podem ser corretos ou não. Os protocolos descritos nesta seção consideram dados genéricos.

3.6.1 Clientes corretos

Aqui, apresentam-se os protocolos para sistemas de quóruns mínimos que não resistem a clientes faltosos.

3.6.1.1 MWMR atômico [34]

Os protocolos de leitura e escrita são executados em um sistema de quóruns mínimos (seção 3.2.3.5). A semântica de consistência é *multi-writer multi-reader* atômica.

Cada servidor mantém as seguintes variáveis locais:

- *listeners*: vetor de clientes que estão lendo no servidor. Para cada posição $listeners[c]$ (referente ao cliente-leitor c) estão os atributos de $listeners[c].v$ e $listeners[c].t$, respectivamente o valor armazenado e o *timestamp* associado a este valor quando do início da leitura de c;

Cada cliente contém as seguintes variáveis locais. Estas variáveis são usadas pelo cliente apenas no protocolo de leitura:

- *largest*: vetor que mantém, em cada posição $largest[s]$, o par $\langle v_s, t_s \rangle$ com o maior *timestamp* t_s recebido de um servidor s. Contém os atributos $largest[s].t$ (o *timestamp*) e $largest[s].v$ (o valor);
- T: conjunto que armazena os $f+1$ maiores *timestamps* do vetor $largest$;
- *answer*: matriz esparsa (preenchida com um valor inicial) que armazena, no máximo, os $f+1$ maiores *timestamps* de cada servidor s de acordo com os elementos armazenados no conjunto T.

Funcionamento da escrita. O protocolo de escrita neste caso funciona de maneira similar ao algoritmo 3 (seção 3.4.1.1), mas com o uso de quóruns assimétricos. Neste caso, entretanto, o cliente envia a sua mensagem UPDATE para todos os servidores do sistema, quando este está em sua configuração mínima ($3f+1$ servidores).

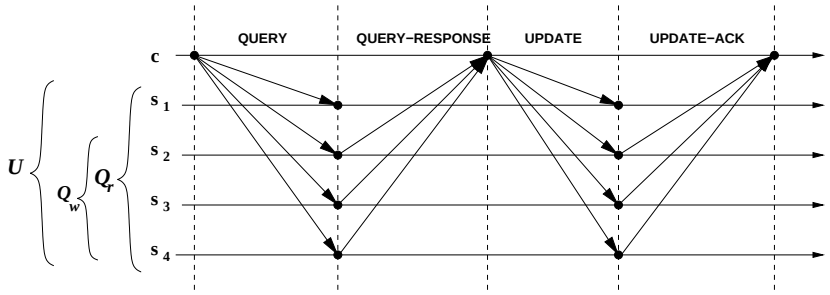


Figura 3.16: Protocolo de escrita – quóruns mínimos, clientes corretos e MWMR atômico para $f = 1$

Funcionamento da leitura (algoritmo 20). O protocolo de leitura possui 2 fases de execução. A primeira fase é semelhante a uma consulta trivial aos pares armazenados em um quórum de leitura: possui uma fase de consulta ao quórum e um passo adicional, no qual o cliente permanece recebendo novas mensagens de servidores até que um quórum de escrita conserve os mesmos valor e *timestamp*. Por esta etapa adicional de comunicação dá-se o nome de padrão de comunicação *listener*, similar ao padrão de projeto de mesmo nome definido em [18].

A presença do padrão *listener* no algoritmo SBQ-L implica, ao mesmo tempo, o número mínimo de $3f + 1$ servidores no sistema (no caso de escritas confirmáveis) e o uso de registradores com semântica atômica. Entretanto, pela própria natureza deste padrão de comunicação, quando leituras e escritas são concorrentes, o número necessário de mensagens para conclusão deste protocolo pode ser maior do que em outros protocolos de leitura já vistos. Em termos práticos, os leitores que utilizam o padrão *listener* acompanham a evolução dos valores escritos durante seus processos de leitura em vez de capturarem apenas um retrato momentâneo (*snapshot*) dos dados na leitura conforme observado nos protocolos anteriores.

Fase 1: o cliente envia uma mensagem QUERY requisitando um conjunto de pares a todos os servidores de um quórum de leitura. Cada par $\langle v_s, t_s \rangle$ recebido de s é armazenado em *largest* se $t_s > largest[s].t$ (linhas 5 a 7). Caso seja a primeira mensagem recebida de um servidor $s \in Q_r$ (linha 8), o cliente atualiza o seu conjunto T com os $f + 1$ maiores *timestamps* em *largest* (linha 10). Em seguida, o cliente atualiza a matriz *answer* para cada servidor, somente permanecendo os registros que estejam em T (linhas 11 a 18). Por fim, o cliente inclui o par $\langle v_s, t_s \rangle$ em *answer* se t_s estiver no conjunto T.

Fase 2: o cliente continua recebendo mensagens de Q_r até que, em *answer*, se encontre um conjunto do tamanho de um quórum de escrita onde todos os pares são iguais ($\langle v_{s'}, t_{s'} \rangle$). Caso esta condição seja satisfeita, o cliente envia uma mensagem READ-COMPLETE para os servidores, encerrando o protocolo de leitura explicitamente (linhas 25 e 26).

Execução do servidor (algoritmo 21). Quando o servidor recebe uma mensagem QUERY de um cliente *c*, aquele verifica se *c* já pertence ao vetor de clientes "ouvintes" (conjunto *listeners*). Se não, põe *c* em *listeners* e depois envia o seu par armazenado por meio de uma mensagem QUERY-RESPONSE.

Quando o servidor recebe uma mensagem UPDATE de um cliente *c*, aquele atualiza seu estado

Algoritmo 20 Leitura de um cliente c

value **function** *read*()
1: $\forall s \in U$, *send*$(s, \langle$QUERY$\rangle)$
2: $S \leftarrow \emptyset$
3: **repeat**
4: $\quad \forall s \in Q_r$,
$\quad$ *receive*$(s, \langle$QUERY-RESPONSE$, \langle v,t \rangle\rangle)$ {é possível o mesmo servidor responder mais de uma vez}
5: $\quad$ **if** $t > largest[s].t$ **then**
6: $\quad\quad largest[s] \leftarrow \langle v,t \rangle$
7: $\quad$ **end if**
8: $\quad$ **if** $s \notin S$ **then**
9: $\quad\quad S \leftarrow S \cup \{s\}$
10: $\quad\quad T \leftarrow largest'$ {largest' tem os $f+1$ maiores *timestamps* de *largest*}
11: $\quad\quad$ **for all** $s' \in S$ **do**
12: $\quad\quad\quad$ **for all** $t' \notin T$ **do**
13: $\quad\quad\quad\quad answer[s',t'] \leftarrow \perp$
14: $\quad\quad\quad$ **end for**
15: $\quad\quad\quad$ **if** $largest[s'].t \in T$ **then**
16: $\quad\quad\quad\quad answer[s', largest[s'].t] \leftarrow largest[s']$
17: $\quad\quad\quad$ **end if**
18: $\quad\quad$ **end for**
19: $\quad$ **end if**
20: $\quad$ **if** $t \in T$ **then**
21: $\quad\quad answer[s,t] \leftarrow \langle v,t \rangle$
22: $\quad$ **end if**
23: **until** $\exists t, v, S', \forall s' \in S' : (|S'| \geq |Q_w|) \wedge answer[s',t] = \langle v,t \rangle$
24: $\forall s \in U$, *send*$(s, \langle$READ-COMPLETE$\rangle)$
25: **return** v
end function

(procedimento *do_update*) se seu *timestamp* armazenado for menor que o *timestamp* recebido do cliente. Em seguida, para todos os clientes $c' \in listeners$, servidor envia uma mensagem QUERY-RESPONSE com o par vindo na mensagem UPDATE caso o *timestamp* do par da mensagem UPDATE for maior que o *timestamp* do par contido em $listeners[c']$. Servidor envia mensagem de confirmação de escrita para o cliente c;

Quando o servidor recebe uma mensagem READ-COMPLETE de um cliente c, aquele retira c do conjunto *listeners*, pois c encerrou o seu procedimento de leitura.

Algoritmo 21 Execução de um servidor s

upon *receive*$(c, \langle$QUERY$\rangle)$
1: **if** $c \notin listeners$ **then**
2: $\quad listeners[c] \leftarrow \langle v_s, t_s \rangle$
3: **end if**
4: *send*$(c, \langle$QUERY-RESPONSE$, \langle v_s, t_s \rangle\rangle)$

upon *receive*$(c, \langle$UPDATE$, \langle v,t \rangle\rangle)$
1: *do_update*$(\langle v,t \rangle)$
2: *send*$(c, \langle$ACK$\rangle)$

upon *receive*$(c, \langle$READ-COMPLETE$\rangle)$
1: $listeners[c] \leftarrow \perp$

procedure *do_update*$(\langle v,t \rangle)$
1: **if** $t > t_s$ **then**
2: $\quad \langle v_s, t_s \rangle \leftarrow \langle v,t \rangle$
3: **end if**
4: **for all** $c' \in listeners$ **do**
5: $\quad t_{list} \leftarrow listeners[c'].t$
6: $\quad$ **if** $t > t_{list}$ **then**
7: $\quad\quad$ *send*$(c', \langle$QUERY-RESPONSE, $\langle v,t \rangle\rangle)$
8: $\quad$ **end if**
9: **end for**
end procedure

Complexidade de mensagens: os algoritmos de leitura, escrita e do servidor possuem comple-

xidade de mensagens de $O(n)$. As operações de escrita e leitura (sem concorrência com escritas), se completam, respectivamente, em 4 e 3 passos de comunicação.

As figuras 3.17 e 3.18 ilustram o funcionamento, respectivamente, da leitura não concorrente à escrita e concorrente à escrita. No caso de concorrência, o cliente leitor c_r inicia sua leitura, e o cliente escritor c_w inicia sua escrita antes de a leitura de c_r terminar.

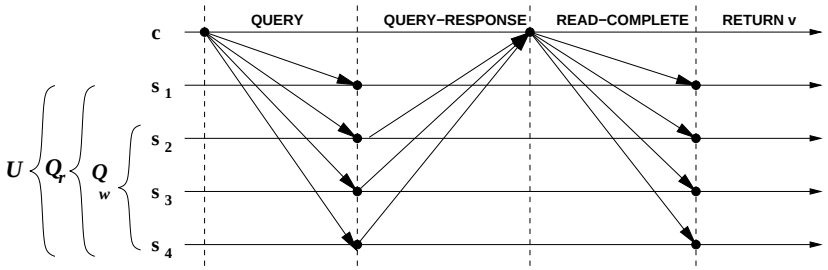


Figura 3.17: Protocolo de leitura – quóruns mínimos, clientes corretos e MWMR atômico sem concorrência para $f = 1$

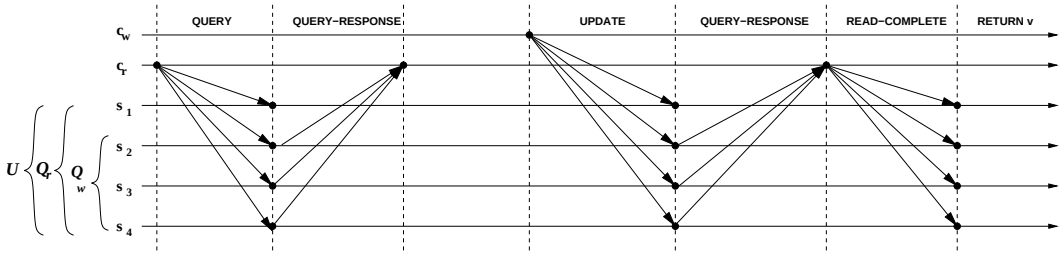


Figura 3.18: Protocolo de leitura – quóruns mínimos, clientes corretos e MWMR atômico com concorrência para $f = 1$

3.6.1.2 MWMR regular [34]

Assim como no caso MWMR atômico (seção 3.6.1.1), os clientes utilizam o algoritmo SBQ-L, porém sem escritas confirmáveis, o que implica uma nova configuração de sistema com um número reduzido de f servidores: $n \geq 2f+1$ servidores com quóruns assimétricos de leitura e escrita respectivamente de tamanhos $|Q_r| = \lceil \frac{n+2f+1}{2} \rceil$ e $|Q_w| = \lceil \frac{n+1}{2} \rceil$, que armazenam registradores com semântica *multi-writer multi-reader* regular.

Enquanto o sistema do caso MWMR atômico utiliza quóruns mínimos para sistemas com escritas confirmáveis, o sistema de quóruns, neste caso, também satisfaz o limite mínimo de servidores para escritas não confirmáveis (mínimo de $2f+1$ servidores e com 3 passos de comunicação na escrita) e não requer dados auto-verificáveis.

3.6.2 Clientes faltosos

Esta seção apresenta protocolos para sistemas de quóruns mínimos tolerantes a clientes faltosos, resultantes de modificações que podem ser feitas nos algoritmos apresentados na seção anterior.

3.6.2.1 MWMR atômico [34]

A configuração do sistema de quóruns neste caso segue as características do sistema no caso MWMR atômico com clientes corretos (seção 3.6.1.1). Entretanto, entre seus protocolos existem algumas diferenças.

A primeira diferença é o uso de assinaturas digitais nos clientes, onde a chave privada é compartilhada entre estes e não é acessível aos servidores, os quais usam a chave pública correspondente. Contudo, esta modificação acarreta um problema: clientes faltosos podem passar a sua chave privada para outros clientes em conluio, os quais, juntos, por exemplo, podem provocar ataques de negação de serviço (*DoS*) no sistema.

A segunda diferença provém da mudança no protocolo de atualização do servidor. Neste caso, utilizam-se procedimentos de verificação das mensagens assinadas pelos clientes e de replicação de mensagens com outros servidores, este último nos casos de atualização efetiva da réplica de um servidor. Estas mudanças visam à manutenção da autenticidade das mensagens vindas dos clientes e à consistência entre os servidores corretos.

Funcionamento da escrita. O protocolo de escrita é idêntico ao exibido na seção 3.6.1.1.

Funcionamento da leitura. No caso trivial, sem concorrência com escritas, a leitura é igual ao descrito na seção 3.6.1.1. Quando existe concorrência com escritas, a leitura considera um novo protocolo do servidor (algoritmo 22).

Execução do servidor (algoritmo 22). O servidor realiza suas operações de maneira similar ao algoritmo do caso com clientes corretos, visto na seção 3.6.1.1. A diferença está quando o servidor recebe uma mensagem UPDATE.

Neste caso, ao receber uma mensagem UPDATE de um cliente c ou de um outro servidor $s' \in U$, que atualizou seu estado pelo procedimento do_update, um servidor s executa o seu procedimento do_update. Tal procedimento verifica inicialmente se mensagem UPDATE é válida. Se assim o for, s atualiza seu estado se o *timestamp* t do par $\langle v,t \rangle$ de UPDATE for maior que seu *timestamp* armazenado t_s.

Em seguida, para cada cliente c' registrado em *listeners*, s envia uma mensagem QUERY-RESPONSE com o par recebido $\langle v,t \rangle$ em UPDATE desde que o *timestamp* t deste par seja maior que o *timestamp* $listeners[c'].t$. Caso esta condição seja satisfeita, s ainda envia $\langle v,t \rangle$ para todos os outros servidores do sistema. Após executar o procedimento do_update, s envia uma mensagem de confirmação caso o emissor de UPDATE seja um cliente.

Algoritmo 22 Execução de um servidor s

upon *receive*$(c, \langle$QUERY$\rangle)$
1: **if** $c \notin listeners$ **then**
2: $\quad listeners[c] \leftarrow \langle v_s, t_s \rangle$
3: **end if**
4: *send*$(c, \langle$QUERY-RESPONSE$, \langle v_s, t_s \rangle\rangle)$

$\{\forall s' \in U, p \in (c, s')\}$

upon *receive*$(p, \langle$UPDATE$, \langle v,t \rangle\rangle_c)$
1: *do_update*$(\langle$UPDATE$, \langle v,t \rangle\rangle_c)$
2: {recebeu UPDATE de um cliente}
3: **if** $p \in \Pi$ **then**
4: $\quad send(p, \langle$ACK$\rangle)$
5: **end if**

upon *receive*$(c, \langle$READ-COMPLETE$\rangle)$
1: $listeners[c] \leftarrow \perp$

procedure *do_update*$(\langle$UPDATE$, \langle v,t \rangle\rangle_c)$
1: **if** *sign_ok*$(\langle$UPDATE$, \langle v,t \rangle\rangle_c)$ **then**
2: $\quad$ **if** $t > t_s$ **then**
3: $\quad\quad \langle v_s, t_s \rangle \leftarrow \langle v,t \rangle$
4: $\quad\quad \forall s' \in U \setminus \{s\}, send(s', \langle$UPDATE$, \langle v,t \rangle\rangle_c)$
5: $\quad$ **end if**
6: $\quad$ **for all** $c' \in listeners$ **do**
7: $\quad\quad t_{list} \leftarrow listeners[c'].t$
8: $\quad\quad$ **if** $t > t_{list}$ **then**
9: $\quad\quad\quad send(c', \langle$QUERY-RESPONSE$, \langle v,t \rangle\rangle)$
10: $\quad\quad$ **end if**
11: $\quad$ **end for**
12: **end if**

end procedure

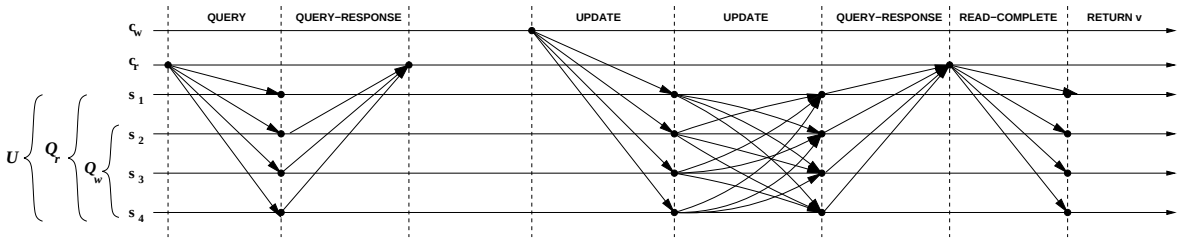


Figura 3.19: Protocolo de leitura – quóruns mínimos, clientes faltosos e MWMR atômico com concorrência para $f = 1$

Complexidade de mensagens: os algoritmos de leitura sem concorrência (algoritmo 20) e escrita (igual ao algoritmo 3 da seção 3.4.1.1) têm complexidade de mensagens de $O(n)$. Quando da concorrência de leituras e escritas, a complexidade da leitura passa a ser de $O(n^2)$ por causa da atualização do servidor (algoritmo 22), que também possui complexidade de mensagens de $O(n^2)$. A escrita e a leitura (sem considerar concorrência com escritas) ocorrem em, respectivamente, 4 e 3 passos de comunicação.

3.6.2.2 MWMR regular [34]

Neste caso, os protocolos de leitura e escrita são implementados para um sistema de quóruns com a mesma configuração do caso MWMR regular com clientes corretos visto na seção 3.6.1.2, usando usando $2f + 1$ servidores e com escritas não confirmáveis. Os protocolos desta seção empregam os procedimentos do caso MWMR atômico com clientes faltosos (seção 3.6.2.1), como o uso de chaves criptográficas pelos clientes e a modificação do protocolo do servidor para manutenção de consistência.

3.7 Discussão e resumo dos algoritmos

Esta seção finaliza a descrição dos algoritmos de Sistemas de Quóruns Bizantinos apresentando primeiramente o histórico de pesquisa na área e, logo em seguida, dois resumos em forma de tabelas dos algoritmos expostos neste capítulo.

3.7.1 Histórico de pesquisa em Sistemas de Quóruns Bizantinos

O uso de sistemas de quóruns como uma solução para se construir sistemas replicados de armazenamento atendendo aos requisitos de disponibilidade de serviço e consistência de dados já decorre de algum tempo [19, 42]. Nestes primeiros modelos de sistema de quóruns, assumia-se apenas a ocorrência de falhas de parada (*crashing*) nos processos, onde um número limitado de réplicas no sistema poderiam apenas parar de funcionar. Deste modo, como requisito de consistência, exigia-se que houvesse sempre servidores corretos em comum a cada dois quóruns do sistema, de maneira que o acesso de leitura a um quórum devolvesse os últimos dados anteriormente escritos em um quórum de escrita.

Malkhi e Reiter [32] introduziram a idéia de sistema de quóruns sob a hipótese de que até um número f de réplicas podem falhar de maneira arbitrária (ou bizantina [26]). O requisito de consistência é garantido pela existência de um número **suficiente** de servidores corretos na interseção entre dois quóruns quaisquer do sistema, enquanto que a disponibilidade do sistema é alcançada pela existência de, pelo menos, um quórum com apenas servidores corretos. Assume-se que processos comunicam-se em canais assíncronos e autenticados.

Neste primeiro trabalho, são apresentados os **sistemas de quóruns de f-mascaramento (*f-masking quorum systems*)**, que requer $4f+1$ servidores com quóruns de tamanhos iguais para leitura e escrita (quóruns simétricos) com $3f+1$ servidores. Duas variantes deste modelo são também apresentadas: os **sistemas de quóruns de f-disseminação (*f-dissemination quorum systems*)**, um caso especial de sistema que utiliza dados auto-verificáveis (dados digitalmente assinados) e, requer, por isso, um número menor de servidores no sistema ($3f+1$ servidores) com quóruns de $2f+1$ servidores; e os **sistemas de quóruns de mascaramento f-opacos (*f-opaque masking quorum systems*)**, nos quais os clientes não conhecem as hipóteses de falhas do sistema. Usando quóruns de f-mascaramento, Malkhi e Reiter descrevem os algoritmos de leitura e escrita em quóruns das seções 3.4.1.1 e 3.4.2.1 considerando, respectivamente, clientes corretos e faltosos; com quóruns de f-disseminação, este mesmo trabalho apresenta os algoritmos descritos na seção 3.4.1.2 considerando clientes corretos.

Em [29], Malkhi e Reiter descreveram o sistema Phalanx, o qual emprega algoritmos melhorados para sistemas de f-disseminação com clientes corretos (seção 3.4.1.3) e de f-mascaramento com clientes faltosos (seção 3.4.2.2). No primeiro caso, com a nova solução, o sistema de quóruns passa a armazenar registradores replicados com semântica atômica — ao invés da semântica regular anteriormente alcançada (seção 3.4.1.2) — utilizando mais um passo de comunicação para reescrita de informações (*write back*) no protocolo de leitura. Já no segundo caso, usando quóruns de f-mascaramento, o novo algoritmo oferece ao sistema uma semântica de acesso para múltiplos escritores — ao contrário da

semântica anterior para um único escritor (seção 3.4.2.1) — empregando o protocolo *Echo Broadcast* [40], de complexidade $O(n)$, em vez da solução anterior com complexidade $O(n^2)$ usando troca de mensagens entre servidores.

Martin et al. [35] propuseram uma nova construção de sistema de quóruns (denominado SBQ, de *Small Byzantine Quorum Systems*) empregando o mesmo modelo de sistema de Malkhi e Reiter, porém com uma redução de f servidores no sistema em relação aos quóruns de f-mascaramento (sistema com $3f+1$ servidores e dados armazenados não assinados) e f-disseminação (sistema com $2f+1$ servidores e dados armazenados auto-verificáveis). Os quóruns de leitura e escrita destes sistemas, denominados, respectivamente, de **a-mascaramento (*a-masking*)** e **a-disseminação (*a-dissemination*)**, possuem tamanhos diferentes (quóruns assimétricos), e somente são considerados clientes corretos. Os protocolos de acesso a estes sistemas (respectivamente, descritos nas seções 3.5.1.1 e 3.5.1.2) não consideram escritas confirmáveis, uma vez que a comunicação do sistema já ocorre em canais confiáveis. Por um lado, o não uso de confirmação favorece uma escrita mais rápida, com garantias de recebimento das atualizações pelos servidores corretos e mantendo as mesmas semânticas de consistência dos casos similares anteriores. Porém, a ausência de confirmação de mensagens pode ser um problema, como nos casos em que um cliente dependa de uma definição local do término da operação de sua escrita para poder realizar uma novo passo.

Neste mesmo trabalho, Martin et al. apresentaram e discutiram a construção de diferentes sistemas de quóruns sobre outros modelos de comunicação, além do modelo com canais assíncronos e confiáveis adotado em [32]. A motivação para isto advém da constatação de que é difícil realizar comunicação confiável sobre um ambiente sabidamente não confiável, ou seja, com premissas de comunicação mais fracas e envolvendo processos bizantinos. Como resultado desta discussão, outros modelos de sistemas de quóruns bizantinos são apresentados, observando-se que, à medida que se enfraquece alguma premissa do modelo de comunicação do sistema (e.g., de síncrono para assíncrono ou de confiável para não confiável), um número adicional de f servidores são necessários para tolerar até f faltas bizantinas. Um dos modelos de sistema de quóruns é o S-SBQ, que expressa sistemas em modelos síncronos usando um parâmetro fixo f (limite de faltas bizantinas) e um parâmetro ajustável t ($t \leq f$, limite de faltas para que operações no sistema sejam executadas sem espera por um *timeout*, não confundir com o t de *threshold* alternativamente usado por alguns autores no lugar do parâmetro f anteriormente citado). No caso de $t = 0$, o modelo S-SBQ torna-se um sistema de quóruns síncrono [5], no caso $t = f$, S-SBQ iguala-se ao próprio sistema com quóruns assimétricos inicialmente proposto (o SBQ). Outros modelos propostos são os sistemas de quóruns *U-masking* e *U-dissemination*, equivalentes, respectivamente, ao *f-masking* e *f-dissemination* em modelos assíncronos não confiáveis. Para ambos os modelos, são definidos também protocolos para quóruns que refletem semânticas de consistência segura (considerando dados armazenados não assinados) e regular (com dados armazenados auto-verificáveis).

Em [34], Martin et. al mostraram que não é possível implementar protocolos com semânticas confirmáveis utilizando menos de $3f+1$ servidores no sistema ou com semânticas não confirmáveis usando menos de $2f+1$ servidores. Este mesmo trabalho ainda apresenta o algoritmo SBQ-L para sistemas com quóruns assimétricos, usando o número ótimo de $3f+1$ servidores para escritas confirmáveis e alcançando uma semântica atômica assumindo clientes corretos (seção 3.6.1.1) ou fal-

tosos (seção 3.6.2.1); e de $2f+1$ servidores para escritas não confirmáveis e alcançando semântica regular com clientes corretos (seção 3.6.1.2) e faltosos (seção 3.6.2.2). Estes quóruns "mínimos", denominação usada neste capítulo para estes sistemas, conseguem armazenar dados genéricos usando um número ótimo de servidores. A novidade do algoritmo SBQ-L está em seu mecanismo de leitura, cujo funcionamento baseia-se no padrão de projeto "listener" [18]. O uso deste padrão implica uma semântica atômica (no caso de escritas confirmáveis) sem uso de reescrita no quórum. Por outro lado, pode ocasionar um número maior de mensagens trocadas quando da leitura e escrita concorrentes se comparado aos protocolos anteriores que oferecem consistências similares.

O trabalho de Liskov e Rodrigues [27] descreve a construção de um registrador atômico que trata uma grande variedade de problemas causados por clientes bizantinos. Este registrador obedece a novas condições de corretude (também apresentadas neste trabalho), mais fortes do que as definidas por Malkhi et al. em [30]. Estas novas condições limitam o número de escritas criadas por clientes maliciosos já retirados do sistema e definem o número necessário de escritas feitas por clientes corretos para sobrescrever possíveis "escritas ocultas" (*lurking writes*) realizadas por clientes faltosos. Como implementação deste registrador, Liskov e Rodrigues apresentam o algoritmo BFT-BC (seção 3.4.2.3), que executa em um sistema de quóruns de f-disseminação (portanto, armazenando dados auto-verificáveis) usando uma fase adicional para preparação de escrita. A fim de garantir as suas fortes condições de corretude, um cliente no BFT-BC utiliza um mecanismo de provas em todas as suas fases de execução. Desta maneira, o protocolo de escrita adiciona mais 2 passos de comunicação em relação aos protocolos que oferecem a mesma consistência embora seja possível utilizar uma versão otimizada do protocolo que mantém o mesmo número de passos dos protocolos anteriores. A semântica de consistência atômica é garantida pelo uso de reescrita no quórum. Porém, este mecanismo de provas envolve o uso de assinaturas digitais, cujo custo computacional é alto.

Bazzi e Ding [6] propuseram algoritmos de sistema de quóruns que estabelecem o uso de *non-skipping timestamps*, cujos valores não crescem arbitrariamente, uma vulnerabilidade que pode ser explorada por clientes bizantinos em grande parte das soluções já propostas. Esta solução, entretanto, requer uma quantidade maior de servidores ($4f+1$) e, para resistir clientes faltosos, assim como no SBQ-L, na escrita, usa assinatura digital e troca de mensagens entre servidores para manter o *timestamp* mais atualizado nos servidores; Cachin e Tessaro [10] apresentam algoritmos para otimizar o armazenamento em sistemas replicados através de fragmentação de dados usando um sistema de quóruns bizantinos com o número ótimo de $3f+1$ servidores, o algoritmo SBQ-L, *non-skipping timestamps* e códigos de apagamento (*erasure codes* [39]). Esta solução garante semântica MWMR atômica e tem boa resistência, mas emprega custosos procedimentos, como assinatura de limiar e difusão confiável entre servidores.

Alguns trabalhos ainda descrevem casos de uso de sistemas de quóruns bizantinos. O sistema Phalanx [29] utiliza BQS para implementar armazenamento de dados tolerante a faltas bizantinas e exclusão mútua (usando quóruns de f-disseminação) tolerante a até f servidores bizantinos. O Fleet [31], outro exemplo de uso de BQS, é um *middleware* em Java para construção de repositório persistente de objetos Java e resistente a f servidores bizantinos. O COCA [48] é uma autoridade certificadora distribuída tolerante a faltas que também usa quóruns de f-disseminação em sua construção. Outro exemplo de uso é o trabalho de Goodson et al. [20] que descreve um protocolo para armazena-

mento confiável usando fragmentação de dados também baseado em sistema de quóruns.

3.7.2 Resumo dos algoritmos

As tabelas a seguir resumem os algoritmos apresentados neste capítulo de duas maneiras: a tabela 3.2 sintetiza as principais características de todos os protocolos de leitura e escrita de sistemas de quóruns bizantinos, relacionando-os às suas seções neste capítulo. A tabela 3.3 categoriza os protocolos por suas semânticas de consistência e de acesso relacionando-os ainda aos seus modelos suportados de falhas nos clientes.

Seção	**Clientes**	**Semântica**	**Servidores**	$\lvert Q_w \rvert$	$\lvert Q_r \rvert$	**Mensagens** [1]	**N^o passos** [2]
3.4.1.1	corretos	segura	$\geq 4f+1$	$\lceil \frac{n+2f+1}{2} \rceil$	$\lceil \frac{n+2f+1}{2} \rceil$	$O(n)$	4 / 2
3.4.1.2	corretos	regular	$\geq 3f+1$	$\lceil \frac{n+f+1}{2} \rceil$	$\lceil \frac{n+f+1}{2} \rceil$	$O(n)$	4 / 2
3.4.1.3	corretos	atômica	$\geq 3f+1$	$\lceil \frac{n+f+1}{2} \rceil$	$\lceil \frac{n+f+1}{2} \rceil$	$O(n)$	4 / 4 [3]
3.4.2.1	faltosos	segura	$\geq 4f+1$	$\lceil \frac{n+2f+1}{2} \rceil$	$\lceil \frac{n+2f+1}{2} \rceil$	$O(n^2)$ $O(n)$	4 / 2
3.4.2.2	faltosos	segura	$\geq 4f+1$	$\lceil \frac{n+2f+1}{2} \rceil$	$\lceil \frac{n+2f+1}{2} \rceil$	$O(n)$	6 / 4
3.4.2.3	faltosos	atômica	$\geq 3f+1$	$\lceil \frac{n+f+1}{2} \rceil$	$\lceil \frac{n+f+1}{2} \rceil$	$O(n)$	6 / 4 [4]
3.5.1.1	corretos	segura	$\geq 3f+1$	$\lceil \frac{n+f+1}{2} \rceil$	$\lceil \frac{n+f+1}{2} \rceil + f$	$O(n)$	3 / 2
3.5.1.2	corretos	regular	$\geq 3f+1$	$\lceil \frac{n+1}{2} \rceil$	$\lceil \frac{n+1}{2} \rceil + f$	$O(n)$	3 / 2
3.6.1.1	corretos	atômica	$\geq 3f+1$	$\lceil \frac{n+f+1}{2} \rceil$	$\lceil \frac{n+3f+1}{2} \rceil$	$O(n)$	4 / 3 [5]
3.6.1.2	corretos	regular	$\geq 2f+1$	$\lceil \frac{n+1}{2} \rceil$	$\lceil \frac{n+2f+1}{2} \rceil$	$O(n)$	3 / 3 [5]
3.6.2.1	faltosos	atômica	$\geq 3f+1$	$\lceil \frac{n+f+1}{2} \rceil$	$\lceil \frac{n+3f+1}{2} \rceil$	$O(n^2)$ $O(n)$	4 / 3 [5]
3.6.2.2	faltosos	regular	$\geq 2f+1$	$\lceil \frac{n+1}{2} \rceil$	$\lceil \frac{n+2f+1}{2} \rceil$	$O(n^2)$ $O(n)$	3 / 3 [5]

Tabela 3.2: Protocolos *versus* características de sistemas de quóruns bizantinos

[1] Complexidade de troca de mensagens na escrita (acima) e na leitura (abaixo).

[2] Número de passos na escrita/número de passos na leitura.

[3] No caso otimizado, são realizados 2 passos na leitura.

[4] Valores para o protocolo normal. No otimizado, são realizados 4 passos na escrita e 2 passos na leitura.

[5] Número de passos de leitura sem concorrência com escritas.

Cliente	**Correto**		**Faltoso**	
Semântica	SW	MW	SW	MW
Seguro	—	3.4.1.1 3.5.1.1 [6]	3.4.2.1	3.4.2.2 —
Regular	—	3.4.1.2 3.5.1.2 [5] 3.6.1.2	—	—
Atômico	—	3.4.1.3 3.6.1.1	—	3.4.2.3 3.6.2.1

Tabela 3.3: Semântica de consistência *versus* natureza de falhas dos clientes *versus* semânticas de leitura e escrita

3.8 Considerações finais

Este capítulo realizou de maneira sistemática uma descrição dos conceitos fundamentais de Sistema de Quóruns Bizantinos e um levantamento dos principais algoritmos para Sistema de Quóruns Bizantinos conhecidos até então. Como resultado, no qual se acredita não possuir similar na literatura, para cada protocolo apresentado, foram apontadas as suas características teóricas (e.g., semânticas de consistência e de acesso, sistema de quóruns empregado, contexto de falhas dos clientes, etc.), bem como as suas descrições algorítmicas das partes cliente e servidor. Ao final do capítulo, traçou-se o histórico de trabalhos em Sistemas de Quóruns Bizantinos, localizando os algoritmos mostrados nos avanços obtidos na área até então, e apresentou-se uma síntese dos algoritmos expostos com suas principais propriedades.

Adotou-se neste capítulo uma notação própria para descrição, uma vez que os trabalhos existentes provêm de diferentes autores, que apresentam inevitavelmente suas abordagens de maneiras diferentes, o que dificulta muitas vezes o entendimento dos conceitos envolvidos e dos protocolos propriamente. Vale ressaltar que, para alguns algoritmos apresentados aqui, não existiam descrições formais, o que aumenta o caráter de contribuição documental deste capítulo.

O capítulo seguinte focalizará no que é considerado o principal objetivo desta dissertação: a implementação do arcabouço para avaliação de algoritmos para BQS.

[6] Protocolo não confirmável.

Capítulo 4

Arcabouço de avaliação de Sistemas de Quóruns Bizantinos BQSNEKO

4.1 Introdução

A existência de um arcabouço de avaliação de protocolos de sistemas de quóruns bizantinos (BQS) surge da necessidade de se analisar diferentes abordagens para a concretização de armazenamento bizantino usando BQS (e.g., [29, 32, 34, 35]). Estas soluções refletem variadas visões de projeto de construção de um sistema de armazenamento, que se diferenciam por um conjunto de requisitos desejados para o sistema, tais como: o nível de consistência de dados suportado e a natureza do acesso dos clientes (clientes realizam suas operações de forma concorrente ou não?); o contexto de falhas do sistema, determinado pela quantidade máxima de servidores bizantinos que podem falhar (número de servidores para que o sistema resista até f faltas) e pela natureza de falhas dos clientes do sistema (bizantinos ou não?).

Para que tal análise seja possível, este arcabouço deve oferecer funcionalidades de implementação rápida dos algoritmos de BQS e de configuração de ambientes bizantinos para execução dos protótipos, a fim de indicar sob quais contextos um algoritmo se desempenha melhor do que outro por exemplo. Assim, decisões de projetos para construção de sistemas de armazenamento usando BQS podem ser tomadas com base nas informações extraídas dos testes com os protótipos antes mesmo da construção propriamente dita do sistema, na intenção de que este, uma vez concretizado, expresse os requisitos iniciais do projeto.

4.1.1 Objetivo e organização do capítulo

Este capítulo descreve o arcabouço para avaliação de sistemas de quóruns bizantinos BQSNEKO, bem como o arcabouço NEKO, base para as soluções realizadas no BQSNEKO. Primeiro, de maneira geral, na seção 4.2, será apresentada a estrutura do arcabouço NEKO, mais especificamente como

se organizam as aplicações neste ambiente. A seção 4.3 exibe a arquitetura do arcabouço BQSNEKO, detalhando as extensões realizadas no NEKO para a implementação de algoritmos de BQS e de perfis maliciosos (para simulação de cenários de ataques). Esta seção ainda contém um exemplo de implementação e configuração de um algoritmo para BQS como forma de ilustração do funcionamento do BQSNEKO. A seção 4.5 fala de alguns trabalhos relacionados ao BQSNEKO. Para concluir o capítulo, a seção 4.6 apresenta as considerações finais.

4.2 NEKO

NEKO [47] é um arcabouço escrito em Java para prototipação e avaliação de algoritmos distribuídos em redes simuladas ou reais. Na arquitetura do NEKO (figura 4.1), um ambiente distribuído de execução organiza-se como um conjunto de processos que se comunicam por passagem de mensagens. Cada processo NEKO mantém uma instância local da aplicação distribuída e executa sobre um ou mais modelos de redes.

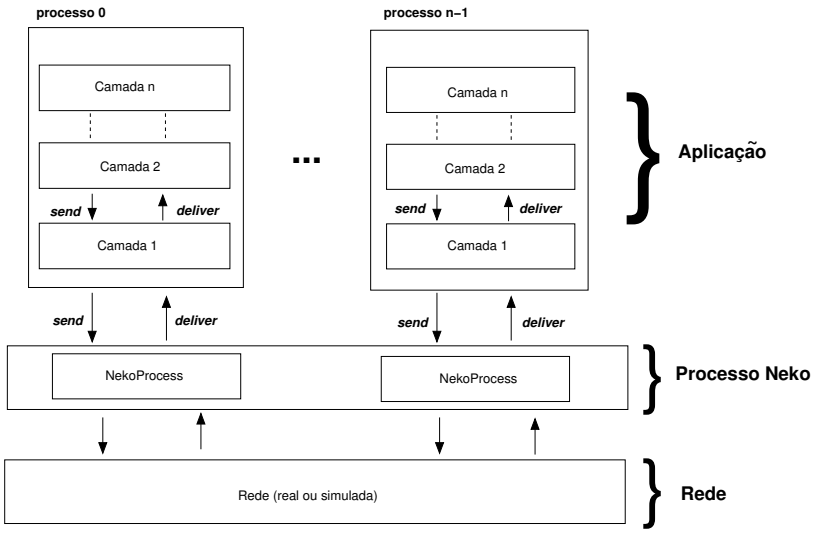
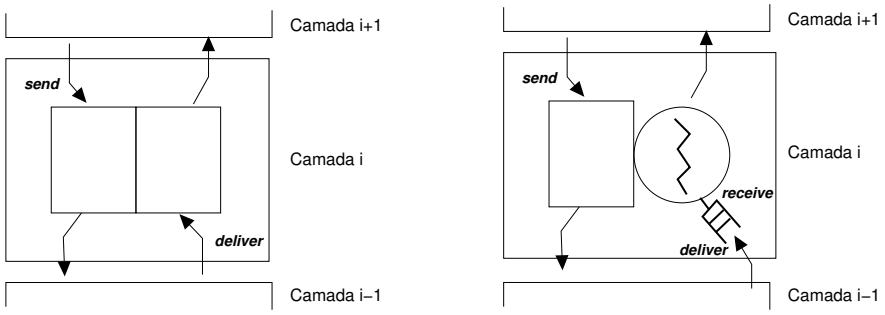


Figura 4.1: Arquitetura do NEKO [47]

Em geral, uma aplicação NEKO organiza-se em camadas[1], onde cada camada oferece um determinado serviço. Camadas comunicam-se trocando mensagens através dos métodos *send* (da camada superior para inferior) e *deliver* (da camada inferior para superior).

Camadas podem ser passivas ou ativas (figura 4.2). Numa camada passiva, mensagens são indiretamente conduzidas pela sua camada inferior usando o método *deliver* conforme mostrado na figura 4.1. Numa camada ativa, em vez do *deliver*, mensagens são diretamente conduzidas usando o método *receive*, que devolve uma mensagem anteriormente recebida e armazenada numa fila de recepção. Esta fila é gerenciada por um processo de controle especial (*thread*). A camada mais inferior da aplicação comunica-se com o processo NEKO que, por sua vez, envia e coleta mensagens da rede. A

[1]Em sua versão mais recente (versão 1.0), o NEKO utiliza um novo modelo de componentes para organizar suas aplicações, não mais através de camadas.



(a) Camada passiva (b) Camada ativa

Figura 4.2: Tipos de camadas de um processo NEKO [47]

rigor, entende-se por mensagens objetos de dados que encapsulam informações sendo trocadas entre aplicações no NEKO, contendo atributos como as identificações do emissor e do receptor, o tipo da mensagem, o conteúdo da mensagem, etc.

O NEKO ainda implementa uma coleção de redes, que permitem a realização de experimentos em situações simuladas ou reais (usando um ou mais computadores). As redes simuladas são implementadas usando uma biblioteca de simulação padrão do próprio arcabouço, enquanto redes reais utilizam Java *sockets* em sua implementação. O NEKO possibilita, ao mesmo tempo, que outras bibliotecas de simulação sejam integradas e novas redes (simuladas ou reais) sejam criadas.

4.3 Arcabouço BQSNEKO

As subseções a seguir descrevem a organização do arcabouço BQSNEKO e as suas funcionalidades oferecidas. De início, são apresentados os aspectos funcionais que motivaram a construção das extensões no NEKO e os requisitos necessários para desenvolvimento de algoritmos para sistema de quóruns bizantinos com o BQSNEKO. Mostra-se também como prototipar com o BQSNEKO: como implementar os algoritmos de BQS e como criar perfis bizantinos para os seus ambientes de execução. Em seguida, é explicado como configurar instâncias de execução para protótipos implementados no arcabouço em questão. Ao final, são ilustrados um exemplo de implementação de um algoritmo simples de BQS no BQSNEKO e uma possível configuração de execução usando este algoritmo implementado.

4.3.1 Arquitetura do BQSNEKO

Considerando a implementação de algoritmos para BQS, o NEKO apresenta, pelo menos, duas limitações. A primeira delas é a ausência de um mecanismo para injeção de faltas bizantinas; a segunda, a ausência de um “esqueleto” para implementação de algoritmos para BQS, que tire proveito das similaridades desta classe de algoritmos. Assim, a fim de um melhor suporte para prototipação

e avaliação de algoritmos para BQS e com vistas ao seu ambiente de execução, o BQSNEKO surge como uma extensão ao *framework* oferecido pelo NEKO.

A arquitetura do BQSNEKO foi desenvolvida de maneira a facilitar a introdução de novos algoritmos para BQS e de novos cenários de ataques com a definição de novos perfis de faltas bizantinas. Basicamente, três aspectos envolvem a implementação de um algoritmo para BQS:

1. **Informações de configuração**: descrevem as características básicas do sistema de quóruns bizantinos usado e os seus parâmetros de configuração (e.g., número de processos no sistema e tamanho dos quóruns de leitura e escrita). Estas informações estão contidas num objeto de dados BQSNEKO e são usadas durante a execução do protocolo em questão;

2. **Mensagens**: conjunto de mensagens usado na comunicação entre processos cliente e servidor no algoritmo implementado;

3. **Protocolos cliente e servidor**: são implementados nos processos do sistema, representando uma aplicação NEKO. Esta aplicação é composta por 4 camadas (3 passivas e 1 ativa) organizadas em 3 níveis, conforme apresentado na Figura 4.3(a):

   (a) **Camada de processo**: camada ativa de um processo genérico de BQS. Os algoritmos cliente e servidor de um sistema de quóruns bizantinos são implementados nesta camada;

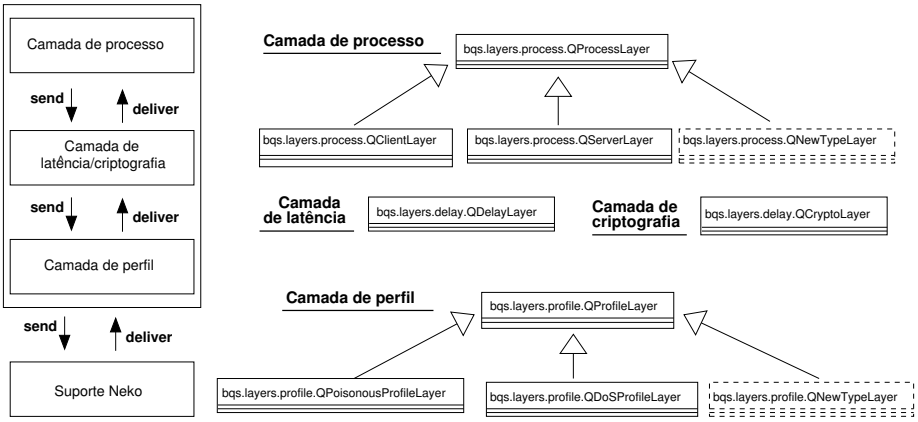
   (b) **Camada de latência/criptografia**: usadas para simular o custo adicional de processamento no envio e recepção das mensagens do protocolo, refletindo o custo de operações criptográficas de execuções em redes simuladas (camada de latência) ou em redes reais (camada de criptografia). Para a camada de latência, o atraso associado ao envio e à recepção de todos os tipos de mensagens do algoritmo pode ser definido como um parâmetro de configuração da sua execução.

   As operações criptográficas são implementadas usando as funcionalidades da biblioteca padrão de criptografia do Java, a JCE (*Java Cryptography Extensions*). A fim de simular o uso de canais autenticados na execução dos protocolos, por padrão, toda mensagem que trafega pela camada de criptografia é atrasada pela simulação de uma operação de *hashing* criptográfico (*Hmac*). Já as operações de assinatura e verificação sofrem um critério de uso de acordo com a descrição do protocolo a ser implementado.

   (c) **Camada de perfil**: define o modelo de falha do processo, ou seja, se a execução é correta (seguindo os algoritmos implementados) ou faltosa (desviando-se arbitrariamente do comportamento esperado).

4.3.2 Prototipando com o BQSNEKO

Implementando um novo algoritmo de um sistema de quóruns bizantinos. Para construir um novo algoritmo de sistemas de quóruns bizantinos, é preciso implementar o objeto de dados BQSNEKO, as mensagens usadas pelos protocolos e os protocolos propriamente.



(a) Arquitetura (b) Classes que implementam as camadas

Figura 4.3: Modelos de camadas de um processo BQSNEKO

Para definir um objeto de dados BQSNEKO e as novas mensagens para os protocolos do sistema implementado, o arcabouço provê duas classes genéricas: uma de objetos de dados e outra de mensagens trocadas em algoritmos de BQS (classes QAbstractInfo e AbstractMessage, respectivamente). A classe QAbstractInfo, que recebe como parâmetro o limite máximo de faltas bizantinas nos servidores de BQS, encapsula informações essenciais para sua construção, tais como o número de servidores no sistema e os tamanhos dos quóruns de leitura e escrita. A classe AbstractMessage mantém informações essenciais para implementação de mensagens de protocolos para BQS, tais como o tipo da mensagem. Assim, a partir destas duas classes, respectivamente, novos objetos de dados específicos e novos tipos de mensagens podem ser criados.

Os protocolos do cliente e do servidor são definidos na camada de processo do BQSNEKO. Esta camada oferece classes genéricas (figura 4.3(b)) do cliente (QClientLayer) e do servidor (QServerLayer). A classe QClientLayer define os métodos abstratos *read* e *write* para operações do cliente. A classe QServerLayer define o método *execute* para execuções do servidor. Estas classes genéricas devem ser estendidas para criar as camadas de processo, que implementam os protocolos cliente e servidor específicos de um determinado sistema de quóruns.

Simulando operações criptográficas (execuções em rede real). Para implementar uma nova mensagem associada ao uso de operações criptográficas de assinatura (no envio) e verificação (na recepção), é preciso que sua classe correspondente estenda AbstractChallengeMessage. Com efeito, tal mensagem implementada terá os custos de assinatura e verificação ativados pela camada de criptografia, respectivamente durante os seus envio e recepção. Para desativar o custo de assinatura (quando se sabe que o emissor não assina a mensagem), use o método *setSignature(boolean)* com o parâmetro igual a *false*. O mesmo pode ser feito com a verificação, nos casos em que o receptor não executa operação uma verificação, usando o método *setVerification(false)*.

Definindo novos perfis de ataques. Para criar um novo perfil de ataque, é preciso primeiro criar um novo perfil bizantino, ou seja, estender a classe genérica QProfileLayer (figura 4.3(b)) da camada

de perfil do BQSNEKO definindo um novo método *send* com o comportamento do processo faltoso. Por exemplo, para implementar um perfil que simule um processo sofrendo um ataque de negação de serviço (DoS), uma idéia seria implementar o método *send* da camada de tal modo que o processo que incorpore este perfil demore *k* vezes mais tempo para realizar sua ação, sendo *k* um parâmetro configurável. Por padrão, a classe genérica da camada de perfil define o método *send* como na execução de um processo correto. A versão atual do BQSNEKO implementa dois perfis de ataque: um perfil venenoso (classe QPoisonousProfileLayer), que altera valores de mensagens, e um perfil bizantino "DoS" (classe QDoSProfileLayer), de acordo com o cenário de ataque descrito anteriormente. Ambas as classes estão ilustradas na figura 4.3(b).

Note que, na prática, as falhas não são injetadas nos processos, mas no canal de comunicação durante o envio da mensagem. Este modelo, onde canais podem corromper (ou omitir) as mensagens arbitrariamente, equivale ao modelo de processos bizantinos, pois todo comportamento malicioso pode ser representado, sendo que suas implementação e configuração em um simulador são muito mais simples.

4.3.3 Executando algoritmos de BQS

Definindo um ambiente de execução. No BQSNEKO, um ambiente de execução de um algoritmo de BQS é definido a partir de um arquivo de configuração estendido do NEKO. Este arquivo está dividido em duas partes: a primeira define as configurações do próprio NEKO, como, por exemplo, se a execução ocorre sobre uma rede simulada ou real, quantos processos existem na execução e qual a classe Java responsável pela iniciação do ambiente de execução (neste caso, a classe BQSInitializer); a segunda parte das configurações diz respeito aos parâmetros específicos dos algoritmos de BQS, como, por exemplo, o número de faltas bizantinas, o tamanho padrão dos quóruns de leitura e escrita, etc. A versão atual do BQSNEKO já implementa os principais algoritmos de sistemas de quóruns bizantinos (tabela 4.1).

Referência	**Quóruns**	**Semântica** [24]	**Clientes**
[32]	simétricos	*MWMR*[1] *safe/regular* *SWMR*[2] *safe*	corretos bizantinos
[29]	simétricos	*MWMR atomic* *MWMR safe*	corretos bizantinos
[34]	assimétricos	*MWMR atomic*	corretos bizantinos
[27]	simétricos	*MWMR atomic*	bizantinos

Tabela 4.1: Alguns algoritmos de BQS implementandos no BQSNEKO

Definindo os protocolos de BQS e suas configurações associadas. Os protocolos cliente e servidor de um sistema de quóruns são definidos pelos parâmetros qclientlayer e qserverlayer, respectivamente.

[1] *multi-writer multi-reader*
[2] *single-writer multi-reader*

O objeto de dados BQSNeko correspondente é definido pelo parâmetro qinfo. O número de servidores faltosos é definido pelo parâmetro faulty.servers.num, e o parâmetro faulty.clients.num designa a quantidade de clientes faltosos para o caso de algoritmos que suportem faltas bizantinas em clientes. O tempo atribuído na camada de latência de um processo (seção 4.3.1) é descrito por um parâmetro no formato latency.<message-type>.<process-type>.event, que define um custo adicional que um processo do tipo <process-type> (*client* ou *server*) terá com uma mensagem de um tipo definido <message-type> durante seu envio (event = *send*) e recepção (event = *receive*). Para conhecer mais parâmetros de configuração, veja o exemplo de configuração da seção 4.4.

Definindo ataques. Em um sistema sujeito a faltas bizantinas, os processos faltosos podem desviar-se da especificação do algoritmo arbitrariamente e assumir outro comportamento qualquer. O BQSNEKO suporta a definição de comportamentos faltosos de maneiras simples e extensível usando o parâmetro de formato faulty.<process-type>.<profile-classname>.percent que define em % a quantidade aproximada de processos do tipo <process-type> (*client* ou *server*) — dentre o número total assumido de processos faltosos do tipo <process-type> — com o perfil faltoso implementado pela classe <profile-classname>. Por exemplo, faulty.server.QDoSProfileLayer.percent = 50 significa que cerca de 50% dos servidores faltosos (arredondando o valor para cima) executam o perfil de faltas implementado pela classe Java QDoSProfileLayer.

Definindo a execução do cliente. A execução do cliente é implementada pelas classes Java TestReadClient e TestWriteClient, para as operações de leitura e escrita no sistema de quóruns, respectivamente. A operação que um cliente de número <X> realiza no sistema de quóruns é definida no arquivo de configuração pelo parâmetro layer.application.type.<X>, onde os seus valores possíveis são *write* (valor padrão) ou *read*. A listagem completa de configuração para operações do cliente é exibida com maiores detalhes na seção 4.4.3.

4.4 Exemplo de implementação e configuração

Esta seção ilustra, de maneira sucinta, a seqüência de passos necessária para a criação de um experimento com algoritmos de BQS usando o BQSNEKO. A seção começa descrevendo a construção do protocolo conforme descrito na seção 4.3.2 e termina com a preparação do arquivo de configuração para execução do protótipo conforme visto na seção 4.3.3. O arquivo de configuração do exemplo ilustra um possível uso dos parâmetros do BQSNEKO mostrando a execução do protocolo de escrita de BQS apresentado na seção 3.4.2.2.

4.4.1 Implementação do protótipo dos protocolos de BQS

Para começar a implementação do protótipo de protocolo de um sistema de quóruns, é necessário definir as informações de configuração do sistema de quóruns usado (e.g., número de servidores no sistema, número de servidores nos quóruns de leitura e escrita, etc.). Assim, o primeiro passo a definir

é o conjunto de informações do sistema de quóruns através da extensão da classe de informações gerais QAbstractInfo. Neste exemplo, a classe de informação do quórum recebe o nome de SymmMWMRFaultySafeInfo (quadro 4.1). A partir de um argumento de limite de faltas f do sistema, esta classe define o número de faltas bizantinas nos servidores, o limite mínimo de servidores no sistema (no caso, $4f+1$) e o tamanho dos quóruns (simétricos, cada um com $3f+1$).

O segundo passo é criar as mensagens usadas pelo protocolo estendendo a classe genérica AbstractMessage (ou, preferencialmente, alguma de suas subclasses abstratas). Use o método *setType(int)* para definir o tipo da mensagem criada (os tipos estão definidos na classe BQSMessageTypes); use o método *setMessage(String)* para definir uma descrição detalhada da mensagem que será escrita em *log*.

Quadro 4.1: Classe de informações do sistema de quóruns

```
1  package lse.neko.applications.bqs.sym.faulty;
2  import lse.neko.applications.bqs.QAbstractInfo;
3
4  public class SymmMWMRFaultySafeInfo extends QAbstractInfo{
5      public SymmMWMRFaultySafeInfo(int numFaults){
6        super(numFaults);
7        //number of servers with fewer 4f+1
8        this.nServers = 4*numFaults + 1;
9        //symmetric quorums with fewer 3f+1 servers
10       this.nReadQuorum = this.nWriteQuorum = 3*numFaults + 1;
11     }
12 }
```

O quadro 4.2 ilustra o código de uma mensagem de consulta ao quórum (*query*) MRQueryMessage, utilizada tanto na leitura quanto na escrita. Esta mensagem é do tipo READ_TS e não carrega um par $\langle v,t \rangle$. Por este motivo, esta mensagem estende a classe AbstractNoObjectMessage, que, genericamente, representa uma mensagem que não tem um par $\langle v,t \rangle$ como atributo.

Quadro 4.2: Mensagem de consulta ao quórum

```
1  package lse.neko.applications.bqs.message;
2  import lse.neko.applications.bqs.BQSMessageTypes;
3
4  public class MRQueryMessage extends AbstractNoObjectMessage{
5     //is query to read?
6     private boolean toRead;
7
8     public MRQueryMessage(boolean toRead, long nonce){
9        setType(BQSMessageTypes.READ_TS);
10       this.nonce = nonce;
11       this.toRead = toRead;
12       String toDoWhat = ((toRead)?" for reading":" for writing");
13       setMessage("read " + toDoWhat);
14    }
15
16    public boolean isRead(){
17       return this.toRead;
18    }
19 }
```

No quadro 4.3, está o código da mensagem MRQueryRespMessage. Esta mensagem é a resposta

do servidor para uma mensagem MRQueryMessage, recebida do cliente durante uma operação de escrita. De maneira correspondente à MRQueryMessage, aquela é do tipo READ_TS. A mensagem MRQueryRespMessage contém um par armazenado $\langle v,t \rangle$ (objeto de leitura-escrita), logo estende a classe genérica AbstractObjectMessage, que representa uma mensagem com um par $\langle v,t \rangle$. Quando da transmissão de MRQueryRespMessage, será escrito em *log* uma mensagem "read value: $\langle v,t \rangle$".

A classe MRQueryRespSignedMessage (quadro 4.4) implementa uma resposta a uma consulta do cliente durante sua leitura. Em conceito, é implementada do mesmo modo que a classe MRQueryRespMessage, exceto pelo fato de representar uma mensagem assinada (tipo READ_TS_SIGN). Esta diferença implica uma pequena, mas importante mudança na implementação: agora, esta estende AbstractChallengeMessage, uma classe genérica que simula, uma mensagem auto-verificável, na qual se verifica a validade de um par $\langle v,t \rangle$ (argumento rwObject).

Quadro 4.3: Mensagem de resposta (na escrita) para uma consulta ao quórum

```
1  package lse.neko.applications.bqs.message;
2
3  import lse.neko.applications.bqs.object.ReadWriteObject;
4  import lse.neko.applications.bqs.BQSMessageTypes;
5
6  public class MRQueryRespMessage extends AbstractObjectMessage{
7     public MRQueryRespMessage(ReadWriteObject rwObject){
8        super(rwObject);
9        //setting a defined message type "read timestamp"
10       setType(BQSMessageTypes.READ_TS);
11       setMessage("read value: "+ rwObject.toString());
12    }
13
14 }
```

Quadro 4.4: Mensagem de resposta (na leitura) para uma consulta ao quórum

```
1  package lse.neko.applications.bqs.message;
2  import lse.neko.applications.bqs.object.ReadWriteObject;
3  import lse.neko.applications.bqs.BQSMessageTypes;
4
5  public class MRQueryRespSignedMessage extends AbstractChallengeMessage{
6
7     public MRQueryRespSignedMessage(ReadWriteObject rwObject, long nonce){
8        super(rwObject);
9        this.nonce = nonce;
10       setType(BQSMessageTypes.READ_TS_SIGN);
11       setMessage("read value: "+ rwObject.toString());
12    }
13
14 }
```

A mensagem de atualização no quórum (quadro 4.5) é, ao mesmo tempo, a requisição do cliente para a escrita e a reescrita. Para o protocolo-exemplo, ela somente está apenas como uma mensagem de escrita. Classifica-se do tipo WRITE. Como resposta à escrita, segundo a implementação do protocolo em ilustração, está o envio pelo servidor de uma mensagem assinada de "eco" do par $\langle v,t \rangle$ recebido do cliente (quadro 4.6). Note que esta mensagem, identificada como do tipo ECHO, estende a classe AbstractChallengeMessage, que, no caso da execução do protocolo em rede, indicará a simulação de uma operação criptográfica durante o envio da mensagem (uma assinatura).

Quadro 4.5: Mensagem de atualização no quórum

```
1  package lse.neko.applications.bqs.message;
2
3  import lse.neko.applications.bqs.object.ReadWriteObject;
4  import lse.neko.applications.bqs.BQSMessageTypes;
5
6  public class MRUpdateMessage extends AbstractObjectMessage{
7
8     private boolean writeBack;
9
10    public MRUpdateMessage(ReadWriteObject rwObj,boolean writeBack,long nonce){
11       super(rwObj);
12       this.writeBack = writeBack;
13       this.nonce = nonce;
14
15       setType(BQSMessageTypes.WRITE);
16
17       String whatOperation = ((writeBack)?" back ":" update ");
18       setMessage("write"+whatOperation+ ": " + rwObj.toString());
19    }
20
21    public boolean isWriteBack(){
22       return this.writeBack;
23    }
24
25 }
```

Quadro 4.6: Mensagem de eco assinado do servidor

```
1  package lse.neko.applications.bqs.message;
2
3  import lse.neko.applications.bqs.object.ReadWriteObject;
4  import lse.neko.applications.bqs.BQSMessageTypes;
5
6  public class MRUpdateEchoSignedMessage extends AbstractChallengeMessage{
7     public MRUpdateEchoSignedMessage(ReadWriteObject rwObj, long nonce){
8        super(rwObj);
9        setType(BQSMessageTypes.ECHO);
10       setMessage("signed echo " + rwObj.toString());
11       this.nonce = nonce;
12    }
13
14 }
```

A classe MRUpdateListMessage (quadro 4.7) implementa a mensagem de escrita pronta (tipo READY). O construtor da classe passa como argumento o par $\langle v,t \rangle$ a ser escrito e o conjunto de ecos assinados recebidos de um quórum (argumento list). Esta classe estende AbstractChallengeMessage, possibilitando, em uma execução do protocolo em rede real, a simulação da operação de verificação das provas assinadas pelo servidor. A classe MRWriteBackListMessage (listagem 4.8) implementa a mensagem de reescrita (*write back*), usada pelo cliente no protocolo de leitura. De maneira similar à implementação da mensagem MRUpdateListMessage, tal classe estende AbstractChallengeMessage no intuito de simular a verificação das provas de reescrita do cliente, visto que, no modelo de falha do protocolo-exemplo, este pode ser bizantino. Por último, a classe MRUpdateAckMessage (quadro 4.9) implementa uma mensagem de confirmação de escrita (ou reescrita) do cliente e estende AbstractNoObjectMessage já que não contém um par $\langle v,t \rangle$.

Quadro 4.7: Mensagem de escrita pronta

```
1  package lse.neko.applications.bqs.message;
2
3  import lse.neko.applications.bqs.object.ReadWriteObject;
4  import lse.neko.applications.bqs.BQSMessageTypes;
5  import java.util.ArrayList;
6
7  public class MRUpdateListMessage extends AbstractChallengeMessage{
8     public MRUpdateListMessage(ArrayList list,ReadWriteObject rwObj,long nonce){
9        super(rwObj,list);
10       setType(BQSMessageTypes.READY);
11       setMessage("write update list: " + rwObj.toString());
12       this.nonce = nonce;
13    }
14 }
```

Quadro 4.8: Mensagem de reescrita (protocolo de leitura)

```
1  package lse.neko.applications.bqs.message;
2  import java.util.ArrayList;
3  import lse.neko.applications.bqs.object.ReadWriteObject;
4  import lse.neko.applications.bqs.BQSMessageTypes;
5
6  public class MRWriteBackListMessage extends AbstractChallengeMessage{
7     public MRWriteBackListMessage(ArrayList list,ReadWriteObject rwObj,long nonce){
8        super(rwObj,list);
9        setType(BQSMessageTypes.WRITE);
10       setMessage("write back list: " + rwObj.toString());
11       this.nonce = nonce;
12    }
13
14    public MRWriteBackListMessage(ArrayList list,ReadWriteObject rwObj,boolean sign, long
          nonce){
15       super(rwObj,list,sign);
16       setType(BQSMessageTypes.WRITE);
17       setMessage("write back list: " + rwObj.toString());
18       this.nonce = nonce;
19    }
20 }
```

Quadro 4.9: Mensagem de confirmação

```
1  package lse.neko.applications.bqs.message;
2  import lse.neko.applications.bqs.BQSMessageTypes;
3
4  public class MRUpdateAckMessage extends AbstractNoObjectMessage{
5     private boolean confirmed;
6
7     public MRUpdateAckMessage(boolean confirmed, long nonce){
8        this.confirmed = confirmed;
9        this.nonce = nonce;
10       setType(BQSMessageTypes.WRITE);
11       setMessage(this.confirmed?"ack":"nack");
12    }
13
14    public boolean isConfirm() {
15       return this.confirmed;
16    }
17 }
```

Depois de implementar as mensagens do protocolo, o terceiro passo é criar as classes dos algoritmos das partes cliente e servidor. O algoritmo do cliente é implementado estendendo a classe genérica QClientLayer. Esta nova classe deve implementar os métodos abstratos *read* (apêndice A) e *write* (apêndice B). O algoritmo do servidor é implementado estendendo a classe genérica QServerLayer. Esta classe deve implementar o método abstrato *execute* (apêndice C)).

4.4.2 Implementando um novo perfil de falta bizantina

Conforme mostrado na seção 4.3.2, para implementar um novo perfil de falta bizantina, é preciso criar uma classe correspondente que estenda a classe genérica QProfileLayer dentro do pacote bqs.layers.profile. Esta nova classe define um novo perfil com um novo identificador (usado para escrita em *log*) e sobrescreve o método *send* da classe QProfileLayer, que implementa o envio de um processo correto. O código no anexo D exibe a implementação da classe QPoisonousProfileLayer, que realiza o perfil venenoso ("pp", de "*poisonous process*"). Note que a corrupção de informação no método *send* só se aplica às mensagens com um par $\langle v,t \rangle$, ou seja, nas mensagens cujas classes estendem AbstractObjectMessage.

4.4.3 Especificando a execução do cliente

A aplicação cliente do sistema de armazenamento, que utiliza uma determinada implementação de registrador, está representada pelas classes TestReadClient (operação de leitura) e TestWriteClient (operação de escrita). A configuração de uma aplicação cliente na execução de um protocolo de BQS fica a cargo de um conjunto de propriedades específicas do arquivo de configuração do BQSNEKO, a saber:

- layer.application.type.<ID>: define de maneira flexível nas configurações de execução de um protocolo o tipo de operação que um processo cliente com identificador igual a <ID> desempenhará. Possíveis valores: *write* (valor padrão) e *read*. Um identificador de processo é definido pelo suporte de execução do NEKO como um valor inteiro entre 0 e $n-1$, onde n é o número total de processos do sistema. Os identificadores de 0 a $k-1$ (k corresponde ao número total de servidores no sistema) são dos servidores; os identificadores de k a $n-1$ são dos clientes.
- layer.application.executions.<operation>: define o número de vezes que um operação do tipo <operation> (*read* ou *write*) será executada. Ao lado da propriedade anterior, esta opção é útil para realização de testes de desempenho de algoritmos. Valor padrão: 1 operação.

4.4.4 Configurando uma execução

O quadro 4.10 mostra um exemplo de configuração que poderia ser usada em um experimento de simulação do algoritmo implementado na seção 4.4.1. A configuração divide-se em duas partes: a primeira, especifica configurações genéricas de NEKO. A segunda, parâmetros específicos do BQSNEKO.

O exemplo exibe uma especificação de simulação do algoritmo sobre uma rede simulada MetricNetwork (linha 8), já implementada pelo NEKO e que tem como parâmetro de entrada $\lambda > 0$ (linha 9). Para fins de explicação do arquivo de configuração, por enquanto não é necessário saber o que é a rede simulada MetricNetwork, tampouco o que significa o seu parâmetro λ.

O sistema tolera, no máximo, 1 falta bizantina nos servidores (linha 18) e define 1 servidor faltoso. Além disso, o experimento define a existência de 2 clientes bizantinos (linha 20). Como o sistema de quóruns neste caso prevê, no mínimo, $4f + 1$ servidores, teremos, portanto, 5 servidores com identificadores que vão de 0 a 4. Adicionando mais 3 clientes (2 deles faltosos), teremos, no total, 8 processos (linha 5). Dos clientes (identificadores de 5 a 7), dois escrevem: os processos com identificadores 6, que é faltoso, e 7, cliente correto (valores padrão, não especificado no arquivo); e um lê: identificador igual a 5 (linha 23), também faltoso. Cada cliente repete sua operação 1000 vezes (linhas 25 e 26). O perfil de falha adotado nos processos faltosos é o mesmo no servidor e nos clientes faltosos, logo 100% das faltas bizantinas dos servidores (linha 19) e clientes (linha 21) são implementadas pela classe QPoisonousProfileLayer, cujo efeito faz com que um processo modifique o valor de suas mensagens enviadas.

Quadro 4.10: Configuração da execução de exemplo

```
1  #### arquivo de exemplo de configuração do BQSNeko ####
2
3  ## 1a parte: configurações do Neko ##
4  simulation = true
5  process.num = 8
6
7  process.initializer = lse.neko.applications.bqs.BQSInitializer
8  network = lse.neko.networks.sim.MetricNetwork
9  network.lambda = 1
10 network.multicast = false
11
12 # parâmetros para registro em log (no arquivo "log.log") da execução do experimentos
13 handlers = java.util.logging.FileHandler,java.util.logging.ConsoleHandler
14 java.util.logging.FileHandler.pattern = log.log
15 messages.level = FINE
16
17 ## 2a parte: configurações do BQSNeko ##
18 faulty.servers.num = 1
19 faulty.server.QPoisonousProfileLayer.percent = 100
20 faulty.clients.num = 2
21 faulty.client.QPoisonousProfileLayer.percent = 100
22
23 layer.application.type.5 = read
24
25 layer.application.executions.write = 1000
26 layer.application.executions.read = 1000
27
28 qinfo = lse.neko.applications.bqs.sym.faulty.SymmFaultyMWMRSafeInfo
29 qclientlayer = lse.neko.applications.bqs.sym.faulty.SymmFaultyMWMRSafeClient
30 qserverlayer = lse.neko.applications.bqs.sym.faulty.SymmFaultyMWMRSafeServer
31 register.type = lse.neko.applications.bqs.object.ReadWriteRegister
```

Como saída (quadro 4.11), tem-se um resumo da configuração do sistema em execução, o tempo médio de execução dos processos clientes e o desvio médio dos mesmos em unidades de tempo de

simulação. Um registro mais detalhado das atividades dos processos fica gravado no arquivo "log.log" conforme atribuído no arquivo de configuração do experimento.

Quadro 4.11: Mensagens de saída da execução de exemplo

```
1  Process (s) 0: pp
2  Process (s) 1: cp
3  Process (s) 2: cp
4  Process (s) 3: cp
5  Process (s) 4: cp
6  Process (c) 5: pp
7  Process (c) 6: pp
8  Process (c) 7: cp
9  Reader p5: average time is 60.0 in 1000 executions and mean deviation of 0.0.
10 Writer p6: average time is 80.54555555555555 in 1000 executions and mean deviation of
       12.92430617283919.
11 Writer p7: average time is 80.54555555555555 in 1000 executions and mean deviation of
       12.942083950616965.
```

4.5 Trabalhos relacionados

O ambiente NEKO [47] compreende um arcabouço que provê algoritmos de consenso, difusão atômica e detecção de faltas, bem como variados modelos de redes reais e simuladas. Estas implementações decorreram de trabalhos que envolveram análise e comparação de algoritmos distribuídos, considerando somente falhas por parada.

Por exemplo, Urbán et al. [46] propuseram métricas para redes com noção de contenção (*contention-aware*) — modelo de rede simulada implementado pelo NEKO que usa um parâmetro λ ($\lambda \geq 0$) de entrada para definir o desempenho relativo entre CPU (processamento local) e rede (transmissão de mensagens), por exemplo $\lambda = 10$ expressaria uma rede local — e utilizaram essas métricas para comparação de algoritmos de difusão atômica; Urbán et al. [44] utilizam o NEKO para avaliação do impacto real da impossibilidade FLP [17] em um ambiente de rede real (uma LAN); Urbán et al. [45] compararam duas importantes abordagens de algoritmos de Consenso observando um conjunto de cenários de falhas por parada em ambiente de rede simulada também usando o NEKO; Ekwall et al. [16] propuseram um novo algoritmo de difusão atômica *token-based* com detectores de faltas e usaram o NEKO para compará-lo com outras duas abordagens de difusão atômica.

No contexto de BQS, embora haja muitas propostas de algoritmos (por exemplo, [27, 29, 32, 34]), poucos trabalhos apresentam análises sobre estes algoritmos: em [34], foi proposto o algoritmo SBQ-L, tolerante a faltas bizantinas, e apresentada uma avaliação deste algoritmo, porém sua análise não considerava a ocorrência de faltas; Goodson et al. [20] compararam uma abordagem de consistência em sistemas replicados de armazenamento bizantino usando BQS com a abordagem baseada em Replicação Máquina de Estados [23, 41], e não considerava também a ocorrência de faltas.

Em relação a ferramentas de simulação de algoritmos distribuídos, é possível atentar a presença de soluções similares ao NEKO (conforme discutido em [47]), mas com distintos focos. Um caso particular é o arcabouço Simmcast-FT [4]: similar ao BQSNEKO (uma extensão também em Java), provê

recursos para simulação de algoritmos distribuídos com injeção de faltas, incluindo faltas bizantinas, mas com foco tão geral como o NEKO (simulação de algoritmos distribuídos). Embora a injeção de faltas do Simmcast-FT tenha mais opções de definição do que o BQSNEKO hoje, o Simmcast-FT, diferentemente do BQSNEKO, não contempla execuções em rede, o que inviabiliza o seu uso (e o de seu mecanismo de injeção de faltas) na experimentação em ambientes reais. Com isto, pode-se dizer que o BQSNEKO consegue reunir um ambiente mais bem integrado e ágil para o seu propósito específico de desenvolvimento e execução de algoritmos de BQS, o que favorece uma análise mais precisa desta classe de protocolos segundo uma gama maior de ambientes de execução. Tal noção evidencia ainda mais as contribuições do presente trabalho.

4.6 Considerações finais

Aproveitando-se da ausência de trabalhos que realizem comparações e análises entre algoritmos de BQS, bem como de uma ferramenta aplicável para tais atividades, o arcabouço BQSNEKO surge como uma solução útil. O BQSNEKO é uma aplicação do *framework* NEKO, cuja infra-estrutura é carente em funcionalidades adequadas para implementação de protocolos pertencentes àquela classe de algoritmos. Tal expediente de avaliação é possível de ser realizado usando o BQSNEKO graças à capacidade do mesmo de explorar características comuns à construção de protocolos de BQS, o que permite as suas implementação e experimentação usando redes reais ou simuladas. Assim, com os resultados obtidos é possível realizar uma posterior avaliação de qual algoritmo adequa-se melhor a determinado ambiente de execução.

Este capítulo apresentou a organização do BQSNEKO e explicou como construir protocolos de BQS e implementar perfis de faltas bizantinas neste arcabouço, além de como configurar uma instância de execução usando estes protocolos relacionados aos perfis de faltas bizantinas previamente criados. Como ilustração, foi mostrado um exemplo de configuração de protocolo para BQS.

Para *download* do BQSNEKO e maiores informações sobre o seu projeto, visite a página em `<http://www.das.ufsc.br/~wagners/bqsneko>`.

Capítulo 5

Avaliação de protocolos de sistemas de quóruns bizantinos com o BQSNeko

5.1 Introdução

Este capítulo mostra como o BQSNEKO pode ser usado para avaliar protocolos de BQS servindo ao propósito de, durante a fase de projeto de um sistema de armazenamento confiável baseado em BQS, auxiliar na indicação de soluções mais adequadas para determinados ambientes de execução. Os resultados dos experimentos descritos neste capítulo, comprovam esta utilidade do arcabouço e levantam também discussões a respeito dos algoritmos experimentados observando suas vantagens e desvantagens quando se consideram determinados ambientes de execução.

Em essência, a avaliação dos protocolos é feita de maneira experimental considerando uma implementação de serviço de armazenamento simples com suporte apenas à leitura e à escrita de dados. A existência desta restrição de operações não é sem motivo, uma vez que um serviço de armazenamento com sistema de quóruns bizantinos consegue, no máximo, implementar uma abstração de registrador atômico [24], sobre o qual é permitido apenas operações de leitura e escrita [22].

A avaliação dos protocolos está distribuída em quatro categorias de análise, cada uma envolvendo a comparação de dois algoritmos com propriedades semelhantes, mas que incorporam em suas implementações técnicas distintas. Nos três primeiros casos, denominados “Métodos de consistência”, “Custo da ‘minimalidade’ ” e “Algoritmos que tratam clientes bizantinos”, avaliam-se somente protocolos de BQS; no quarto caso, avalia-se o custo de armazenamento tolerante a faltas bizantinas entre um protocolo de sistema de quóruns e uma técnica similar baseada em Replicação Máquinas de Estados [23, 41].

A preocupação principal das análises realizadas é observar o comportamento dos algoritmos em um ambiente de rede local, pois se admite aqui a dificuldade de se organizar e administrar um experimento de um serviço de armazenamento em uma rede de larga escala. Em alguns casos, contudo, procura-se estabelecer relações e projeções dos resultados obtidos no modelo de rede local com alguns

testes adjacentes, em um ambiente simulado de rede de larga escala. Em todos os casos, consideram-se apenas sistemas com um número mínimo de réplicas, devido ao alto custo de se implementar independência de falhas em um serviço com uma grande quantidade de réplicas [37].

O capítulo corrente organiza-se da seguinte forma: a seção 5.2 descreve os aspectos de configuração dos experimentos; a seção 5.3 apresenta propriamente os casos de avaliação dos algoritmos discorrendo as abordagens teóricas percebidas na literatura e constratando-as com os resultados obtidos nos experimentos efetuados. A seção 5.4 encerra o capítulo apresentando as reflexões finais.

5.2 Configuração dos experimentos

Os experimentos apresentados neste capítulo foram realizados sobre diferentes ambientes de execução, cujas configurações variam basicamente na condição de carga dos servidores (processos executando concorrentemente ou não) e no número de faltas no sistema (número de servidores bizantinos). Estes aspectos são descritos pormenorizadamente a seguir.

5.2.1 Ambiente de rede

Nas execuções, considera-se uma infra-estrutura de rede local com canais confiáveis e autenticados do sistema usando *sockets* TCP, implementado pelo NEKO, e chaves de sessão baseadas no algoritmo *HmacSHA-1*, implementado pela camada de criptografia do BQSNEKO. Além da autenticação do canal, comum ao modelo do sistema de quóruns bizantinos, alguns protocolos avaliados fazem o uso de criptografia assimétrica. Esta funcionalidade é implementada pelo BQSNEKO por meio de sua camada de criptografia, que emprega o esquema de assinatura com os algoritmos *SHA-1* e *RSA* (1024 bits) para resumos e assinaturas, respectivamente.

A rede local consta de máquinas com a mesma configuração de *hardware* (AMD Athlon XP 1.9Ghz, 512MB de RAM, placa *ethernet* de 100MB/s) interconectadas por um *switch* 1GB/s. O ambiente de *software* em todas as máquinas é também homogêneo: sistema operacional Linux, *kernel* 2.6.12, e máquina virtual Java da SUN versão 1.5.0_06. Por limitação de recursos computacionais disponíveis, serão utilizadas no total 5 máquinas.

Para os casos sem concorrência, serão feitas projeções sobre o desempenho dos algoritmos experimentados a partir de testes executados em um **ambiente de rede simulado**. Para execução em rede simulada, adota-se o **Modelo de Rede com Noção de Contenção (*contention-aware simulated network model*)** [46], oferecido pelo NEKO e inspirado no modelo de Rede Ethernet definido em [43]. Tal modelo de rede, que já foi aplicado em alguns trabalhos de análise de protocolos como [16], [45] e [46], leva em conta o efeito da contenção de recursos no processamento local e na rede, o que habilita uma análise mais precisa dos protocolos. Neste caso, representa-se a contenção por um parâmetro λ ($\lambda \geq 0$), que especifica o desempenho relativo entre recursos de processamento local e a rede. Processos comunicam-se nesta rede usando mensagens ponto-a-ponto (*unicast*), isto é, processos têm um custo de rede para cada mensagem transportada.

Normalmente, para redes locais (LANs), utiliza-se $\lambda > 1$ (custo de processamento local maior que o de rede); $\lambda < 1$ representa modelos de rede com maior contenção no canal de comunicação e $\lambda = 1$ define modelos em que não há diferença entre as contenções nos recursos locais e de rede. Aqui, nos casos de simulação, são utilizados valores de $\lambda = 0.1$ para representar uma rede de larga escala (seguindo trabalhos similares como [45]). Isto representa um ambiente de rede que se caracteriza por uma contenção de recursos diferenciada, onde o tempo para transmissão de dados é normalmente maior do que o tempo para processamento local, o que leva a um custo de processamento praticamente desprezível. Sendo assim, não se considera custo com criptografia, apenas o custo com a transmissão de mensagens.

5.2.2 Métricas

Considera-se o **tempo de latência** como o tempo para terminação de um algoritmo. Neste caso, é o tempo gasto por um processo (consideramos aqui somente clientes corretos) para realizar uma operação de leitura ou escrita, do passo inicial do protocolo (mesmo antes do envio da primeira mensagem) ao último passo necessário para o término do protocolo, embora o processo possa ocasionalmente receber mensagens depois da sua conclusão. Já as **mensagens extras** é a quantidade de mensagens adicionais enviadas ou recebidas da rede por um processo correto durante a sua execução, quando este processo confronta-se com servidores bizantinos. O propósito de medir mensagens extras é dar uma idéia do impacto gerado no desempenho de clientes corretos por processos bizantinos ou por concorrência de operações de acordo com os modelos de falha adotados em cada cenário de avaliação.

Todos os valores depreendidos dos testes em rede real representam o tempo médio necessário (em milisegundos), juntamente com o seu desvio médio, para a execução de uma operação por um cliente do sistema, recolhido a partir de 1000 repetições. Em alguns casos, coleta-se também o número médio de mensagens extras gerados pelo cliente no protocolo. No caso dos testes em rede simulada, que se apresenta como um caso aproximado da sua contraparte real, os valores de tempo (em u.t.s., unidades de tempo simulado) são recolhidos a partir da execução de uma única operação do cliente.

5.2.3 Carga de faltas e outras características do sistema

Para a execução dos experimentos, consideram-se sistemas resistentes a um limite de faltas t, para os valores $t = 1$ e $t = 2$. Em cada caso de limite de falta, admite-se ora um sistema com todos os servidores corretos ($f = 0$), ora com um número de servidores faltosos menor do que o limite de faltas (por exemplo, para $t = 2$ e $f = 1$), ora com todos os servidores faltosos ($f = t$). Considera-se somente um único tipo de falta no sistema durante os experimentos: um processo pode forjar um valor de mensagem antes de enviá-la. Os protocolos sempre executam em sistemas de quórum com um limite mínimo de servidores. O tipo de registrador implementado pelo sistema varia de acordo com cada algoritmo presente nos casos de avaliação, cujo enfoque procura comparar protocolos e sistemas de quóruns com propriedades similares.

5.3 Casos de avaliação

Esta seção apresenta os casos de avaliação de algoritmos de BQS. Doravante, usaremos a seguinte nomenclatura para identificar os algoritmos para BQS em análise: PHALANX (seção 3.4.1.3), MWMR-SEGURO (seção 3.4.2.2), SWMR-SEGURO (seção 3.4.2.1), BFT-BC (seção 3.4.2.3), MINIMAL-CORRETO (seção 3.6.1.1) e MINIMAL-FALTOSO (seção 3.6.2.1).

Os casos de avaliação apresentados aqui subdividem-se em 4 categorias:

(a) **Métodos de consistência** (seção 5.3.1): avaliação dos mecanismos de consistência utilizados nos algoritmos SWMR-SEGURO e MWMR-SEGURO, que implementam registradores com semântica segura e são resistentes a clientes bizantinos;

(b) **Custo da "minimalidade"** (seção 5.3.2): avaliação do custo decorrente do uso de um "registrador atômico mínimo" não tolerante a clientes faltosos, mantido em um sistema de quóruns com o limite mínimo teórico de réplicas de $3f+1$ [34] e implementado pelo protocolo MINIMAL-CORRETO. O custo deste registrador mínimo é comparado com o custo de um registrador implementado pelo protocolo PHALANX, que oferece também um registrador atômico não tolerante a falta nos clientes;

(c) **Algoritmos que tratam clientes bizantinos** (seção 5.3.3): avaliação das técnicas empregadas pelos algoritmos BFT-BC e MINIMAL-FALTOSO para alcançarem registradores com semânticas atômicas de armazenamento em um ambiente passível a faltas bizantinas tanto nos servidores como nos clientes;

(d) **Analisando o custo de armazenamento: BQS** *X* **Paxos** (seção 5.3.4): avaliação dos custos oriundos da construção de sistemas de armazenamento tolerantes a faltas bizantinas, empregando uma abordagem de Sistemas de Quóruns [32] (algoritmo BFT-BC [27]) e uma abordagem de Replicação Máquina de Estados [23, 41] (algoritmo PAXOS [11, 25]).

5.3.1 Métodos de consistência

No modelo de falhas de um sistema de quóruns, clientes também podem ser bizantinos. Nesta situação, o cliente faltoso é capaz de, por exemplo, escrever valores diferentes nos registradores mantidos pelo sistema, de forma que dois servidores corretos não retornem o mesmo valor quando solicitados em uma operação simples de leitura sem concorrência. Para resistir a casos como este, é necessário que o serviço de armazenamento lance mão de algum mecanismo especial em seus protocolos de acesso (geralmente, nas operações de escrita, visto que esta altera o estado do sistema), a fim de que as propriedades de consistência dos registradores associados aos servidores corretos mantenham-se intactas mesmo com a possível atuação de clientes maliciosos no sistema.

Em [32], é apresentada a primeira versão de um protocolo de escrita tendo em vista a possibilidade de clientes bizantinos infrigirem a propriedade de consistência do sistema de quóruns. Esta solução,

denominada aqui de SWMR-SEGURO, possibilita a construção de um sistema que mantém registradores com semântica de consistência segura e emprega, no protocolo de escrita, a difusão confiável de mensagens entre os servidores para assegurar que um valor escrito em um servidor correto seja escrito em todos os outros servidores corretos. O protocolo no SWMR-SEGURO, contudo, é bastante simples: na leitura, não suporta clientes faltosos; na escrita, não suporta a semântica "vários escritores" (*multi-writer*). Os requisitos mínimos de sistema são quóruns simétricos de tamanho $3f+1$ e um sistema com pelo menos $4f+1$ servidores.

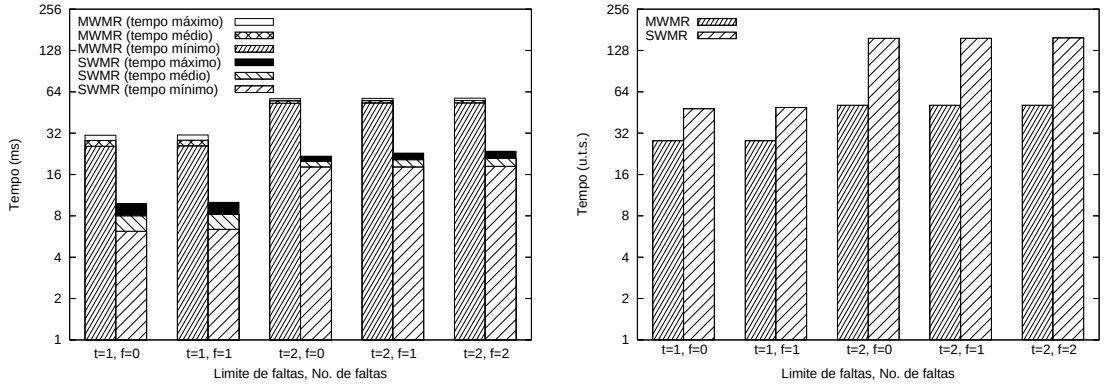
Mais tarde, em [29], uma nova solução, que utiliza as mesmas características de quórum do SWMR-SEGURO, é apresentada. Entretanto, esta solução aprimorada, chamada aqui de MWMR-SEGURO, usa o protocolo de difusão com eco apresentado em [40] para evitar que um cliente malicioso escreva diferentes valores em diferentes servidores corretos. Tal mecanismo de consistência requer o uso de assinaturas por parte dos servidores e um maior número de passos de execução do que a solução anterior (6 no MWMR-SEGURO contra 4 no SWMR-SEGURO). Ao contrário do SWMR-SEGURO, o MWMR-SEGURO suporta a semântica "vários escritores" na escrita e clientes faltosos em operações de leitura, o que implica, neste último caso, também o uso de assinatura pelos servidores.

Este caso de experimento compara os custos associados ao uso das técnicas de consistência no SWMR-SEGURO e MWMR-SEGURO, respectivamente, a difusão confiável entre servidores e o uso de difusão com eco. O objetivo aqui se restringe somente a avaliar estes dois algoritmos em suas operações de escrita sem concorrência, não se preocupando em discutir os protocolos de leitura de ambos, uma vez que um deles não tolera clientes faltosos. Tampouco não se preocupa aqui a avaliação dos casos de concorrência, uma vez que o SWMR-SEGURO não oferece possibilidade de acesso "vários escritores" na escrita.

A figura 5.1 mostra os tempos de execução resultantes das operações de escrita sem concorrência de um cliente em ambientes de rede local e de larga escala, este último simulado. A figura 5.1(a), que mostra resultados em uma LAN, aponta um melhor desempenho do protocolo SWMR-SEGURO em relação ao MWMR-SEGURO em todos os contextos de falta experimentados. Tal cenário se justifica, porque, em uma rede local, há normalmente um maior custo com processamento local no caso do protocolo MWMR-SEGURO usando assinaturas para o seu mecanismo de difusão com eco, em vez das trocas de mensagens entre servidores do SWMR-SEGURO. Vale observar que, mesmo com a presença de servidores faltosos, para o MWMR-SEGURO em rede local, as latências obtidas foram bastante próximas (mesmo fora do desvio médio), uma vez que, do ponto de vista do cliente, não há diferença se as informações recebidas foram de um servidor correto ou faltoso (as verificações criptográficas somente são feitas do lado servidor); para o SWMR-SEGURO, os tempos de execução também foram próximos porque o tempo para troca de mensagens neste caso é irrisório, ainda que ocorram faltas em alguns servidores.

No caso das execuções no modelo de rede de larga escala (figura 5.1(b)), inverte-se a situação do desempenho dos protocolos em questão. Agora, tem-se um contexto distinto de contenção no modelo de rede, onde o tempo de comunicação é normalmente maior do que o tempo de processamento local. Isto se reflete com um impacto maior no mecanismo de difusão confiável entre servidores em relação ao uso de assinatura nos mesmos. Assim, os valores obtidos do MWMR-SEGURO foram

os mesmos na simulação (latência com criptografia torna-se irrisória neste caso, porém, na prática, esperam-se valores próximos de tempo); no SWMR-SEGURO, os dados coletados apresentam pequenas diferenças, embora estas não sejam perceptíveis em gráfico: para $t = 1$, obtiveram-se $48,2$ u.t.s. ($f = 0$) e $49,2$ u.t.s. $f = 1$; para $t = 2$, obtiveram-se $157,2$ u.t.s. ($f = 0$ e $f = 1$) e $158,2$ u.t.s. ($f = 2$). Esta diferença de valores de tempo no SWMR-SEGURO aponta um efeito colateral da troca adicional de mensagens entre servidores em uma rede de larga escala nos casos de falta no sistema, antes imperceptível no caso de rede local.



(a) Escrita em rede local.

(b) Escrita em rede simulada, $\lambda = 0.1$.

Figura 5.1: Desempenho da escrita: MWMR-SEGURO e SWMR-SEGURO (sem concorrência)

5.3.2 Custo da "minimalidade"

Construir protocolos que implementem um registrador atômico não é uma tarefa simples. Tal dificuldade existe porque o protocolo a ser desenvolvido precisa garantir que, até ocorrer uma nova escrita, todos os valores obtidos por operações de leitura têm que ser o mesmo, isto é, o último valor escrito no sistema. Se ocorrem apenas escritas não concorrentes, esta tarefa torna-se mais fácil de ser concretizada. Caso contrário, com escritas concorrentes à leitura, é necessário o uso de mecanismos não triviais para se alcançar a propriedade atômica.

A primeira construção algorítimica utilizada para se atingir a "atomicidade" em protocolos de BQS foi o mecanismo de reescrita (*write-back*), presente no protocolo de leitura do sistema PHALANX [29]. Este mecanismo adiciona um passo facultativo de comunicação na leitura por intermédio de um acesso extra a alguns servidores do sistema, onde o cliente, antes de terminar a sua operação, escreve de volta o par $\langle v,t \rangle$ lido somente nos servidores que não enviaram $\langle v,t \rangle$. Este acesso extra garante que um quórum de leitura conterá $\langle v,t \rangle$ e que, com efeito, todas as leituras subseqüentes (antes de uma nova escrita) resultarão no mesmo valor v. Assim, para que se atenda a propriedade atômica nos registradores do sistema, acrescentam-se mais dois passos no protocolo de leitura do cliente. O algoritmo PHALANX usa um sistema de quórum de f-disseminação (seção 3.2.3.2), que requer quóruns simétricos e, no mínimo, $3f + 1$ servidores no sistema. Neste caso, o registrador atômico não tolera clientes faltosos.

Mais adiante, em [34], provou-se a impossibilidade de construção de sistemas de armazenamento tolerantes a f faltas bizantinas com menos de $3f+1$ servidores implementando registradores com o mínimo de consistência possível (semântica segura) e usando protocolos de escrita confirmáveis. Este trabalho apresenta o algoritmo SBQ-L (MINIMAL-CORRETO na nomenclatura deste capítulo), que implementa um registrador atômico com resiliência ótima, ou seja, capaz de ser construído em um sistema de quóruns com o limite mínimo de $3f+1$ servidores no sistema usando protocolo de escrita confirmável. O algoritmo MINIMAL-CORRETO emprega um sistema com quóruns assimétricos (quóruns de escrita menores que os quóruns de leitura) e um mecanismo distinto para obter atomicidade, baseado no padrão de projeto *listener* [18]. No padrão *listener*, um leitor, quando efetua uma consulta em um quórum de leitura, se registra como *listener* nos servidores deste quórum. Em caso de concorrência com escritas, o leitor registrado recebe os valores de réplicas ocasionalmente atualizadas até que algum valor seja consolidado em um quórum de escrita, isto é, retornado por, pelo menos, $2f+1$ servidores. Em seguida, o leitor solicita aos servidores para que seu registro como *listener* seja cancelado. Assim como no PHALANX, o protocolo MINIMAL-CORRETO não tolera clientes faltosos.

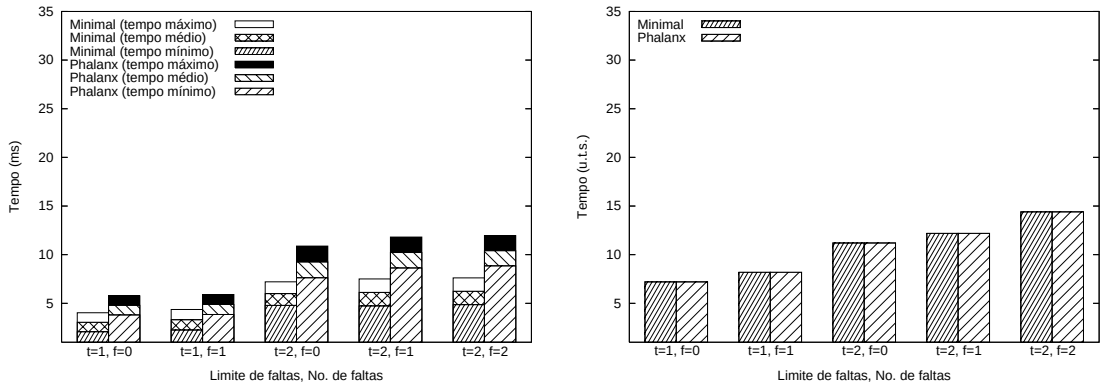
O objetivo deste caso de experimento é avaliar o custo de "minimalidade", ou seja o custo envolvido na construção de um registrador atômico aplicando o padrão *listener* (algoritmo MINIMAL-CORRETO) em comparação à abordagem de reescrita ou *write-back* (algoritmo PHALANX). Além da avaliação pelo tempo de latência, este caso apresentará outro parâmetro de avaliação: a redundância de mensagens manifestada pelo uso de ambas as técnicas em situação de concorrência de operações de leitura e escrita. As mensagens extras neste caso correspondem ao número de vezes em que ocorreram tanto mensagens de reescrita geradas por um leitor no algoritmo PHALANX como mensagens adicionais coletadas por um cliente em sua leitura durante a execução do algoritmo SBQ-L, que usa o padrão *listener*.

A figura 5.2 apresenta os resultados de leituras sem concorrência com outras operações em ambientes de rede local e de larga escala simulado. Na figura 5.2(a), temos ilustrados os experimentos em rede local: em todas as situações de falta, considerando somente os valores médios de latência, temos pequenas diferenças de desempenho a favor do MINIMAL-CORRETO em relação ao PHALANX. Esta diferença ocorre devido ao uso de criptografia pelo cliente do PHALANX na operação de leitura, quando este verifica se cada par $\langle v,t\rangle$ (dado auto-verificável) recebido do quórum não foi modificado por um servidor bizantino. A diferença observada entre ambos os protocolos é pequena, pois a verificação não é muito custosa, ao contrário da assinatura.

Neste caso, não temos operação de reescrita, visto que não há concorrência com operações de escrita, logo os dados retornados pelos servidores corretos (dados corretamente assinados), utilizados para selecionar $\langle v,t\rangle$, são os mesmos. Já no MINIMAL-CORRETO, não temos este processo de verificação criptográfica, o que justifica uma latência menor de execução no seu procedimento de leitura. Em média, o impacto dos servidores bizantinos em ambos os casos é normalmente baixo: para $t=1$, há um aumento de latência de $\approx 2,1\%$ no PHALANX e de $\approx 8,2\%$ no MINIMAL-CORRETO; para $t=2$, no PHALANX, o aumento é de $\approx 13,1\%$, no MINIMAL-CORRETO, de 4%.

Na simulação em rede de larga escala (figura 5.2(b)), o desempenho dos dois protocolos é igual, indicando uma diminuição (ou, talvez, um completo desaparecimento) do efeito da verificação crip-

tográfica no protocolo de leitura do PHALANX, antes notável durante os testes em rede local. Embora a leitura do MINIMAL-CORRETO execute em 3 passos, sua última etapa é descartável na contabilização da latência de execução total, pois a mensagem de cancelamento de inscrição como *listener* não requer resposta. Como resultado, a latência de leitura no MINIMAL-CORRETO aproxima-se da leitura no PHALANX, que executa em 2 passos (não há reescrita).



(a) Leitura em rede local.

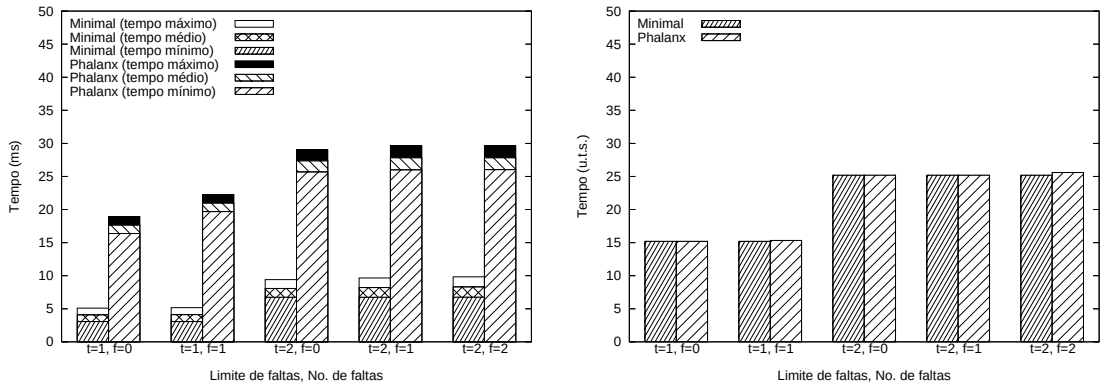
(b) Leitura em rede simulada, $\lambda = 0.1$.

Figura 5.2: Desempenho da leitura: MINIMAL-CORRETO *X* PHALANX (sem concorrência)

A figura 5.3 exibe os resultados para a escrita sem concorrência. Comparando-se os casos de experimentos em rede local (figura 5.3(a)) e rede de larga escala simulada (figura 5.3(b)), fica claro o custo com o uso da criptografia no PHALANX. O esqueleto de funcionamento dos dois protocolos é idêntico: cada qual consulta dados no quórum, cria um novo par $\langle v,t \rangle$, tenta escrever em um quórum e espera um conjunto de confirmações dos servidores. Entretanto, no PHALANX, a fim de tornar o dado armazenado auto-verificável, para cada par $\langle v,t \rangle$ a ser escrito, o cliente realiza uma operação de assinatura usando o algoritmo *RSA*, cujo tempo de processamento é de aproximadamente 14 ms em nosso ambiente de execução.

A presença deste passo com assinatura na execução do PHALANX em LAN é notável em seu desempenho, o que possivelmente não ocorreria em rede de larga escala, cujo custo com processamento local, a priori, é menor do que o custo com a comunicação entre processos. Na simulação em rede de larga escala ilustrada na figura 5.3(b), isto é demonstrado desprezando o custo com processamento local (incluindo a assinatura) e considerando apenas os passos de comunicação. Como resultado, os valores apresentados no gráficos são rigorosamente iguais. Tais valores tenderiam a ser iguais, no entanto, caso as execuções fossem feitas propriamente sobre uma rede de larga escala.

A figura 5.4 exibe os resultados de um escritor concorrendo com leitores, variando de 1 a 5 clientes. Em todos os casos, o MINIMAL-CORRETO manteve uma latência menor do que o PHALANX. A figura 5.5 mostra resultados da execução de um leitor concorrendo com um escritor e outros leitores (variando de 0 a 4) em cenários com limites de falta iguais a 1 (figura 5.5(a)) e 2 (figura 5.5(b)). Considerando apenas os valores médios em cada caso de carga no sistema (faltas e concorrência), percebem-se pequenas mudanças de comportamento no desempenho do leitor no



(a) Escrita em rede local.

(b) Escrita em rede simulada, $\lambda = 0.1$.

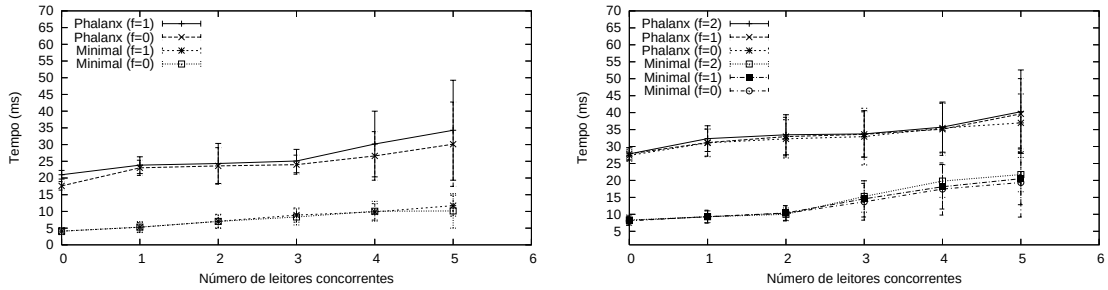
Figura 5.3: Desempenho da escrita: MINIMAL-CORRETO *X* PHALANX (sem concorrência)

MINIMAL-CORRETO. Enquanto isto, a leitura no PHALANX indica um crescimento mais acentuado a partir de 4 leitores concorrentes, sobretudo quando $t = 2$.

A tabela 5.1 exibe os percentuais relativos ao número de vezes em que se realizaram leituras com reescritas no PHALANX e com o uso do padrão *listener* no MINIMAL-CORRETO. Tais valores revelam que, no PHALANX, no pior caso e considerando uma rede local, a concorrência de operações de leitura com escrita ocorre em quase 7% das vezes para $t = 1$ e de 11% para $t = 2$, indicando um baixo uso do mecanismo de reescrita. No protocolo MINIMAL-CORRETO, observando as execuções do leitor com pior desempenho para cada contexto de faltas experimentado, a concorrência mostrou-se mais intensa do que no PHALANX sob um ponto de vista de mensagens extras. Foram realizadas leituras no MINIMAL-CORRETO com o uso de mensagens adicionais no pior caso em $88,46\%$ das vezes para $t = 1$. Para $t = 2$, em quase todas as leituras, existiu a percepção de concorrência: em $99,24\%$ das execuções. Portanto, ainda que, em termos de latência, tal como mostrado em [34] e neste experimento, não seja grande a variação de desempenho de uma leitura no MINIMAL-CORRETO em situação de concorrência, isto não significa uma fraca concorrência no que concerne à geração de notificações extras pelo padrão *listener*.

Protocolo / faltas	$t = 1$		$t = 2$		
	$f = 0$	$f = 1$	$f = 0$	$f = 1$	$f = 2$
MINIMAL-CORRETO	$88,46\%$	$81,68\%$	$94,38\%$	$96,9\%$	$99,24\%$
PHALANX	$6,84\%$	$5,98\%$	$10,84\%$	$9,92\%$	$7,88\%$

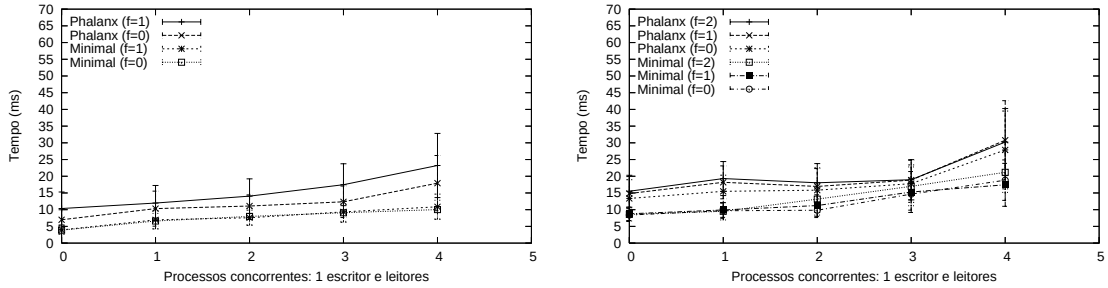
Tabela 5.1: Percentual de leituras com uso do padrão *listener* (MINIMAL-CORRETO) e de reescritas (PHALANX) – concorrência com 1 escritor e leitores



(a) Escrita para t = 1.

(b) Escrita para t = 2.

Figura 5.4: Desempenho da escrita em rede local: MINIMAL-CORRETO *X* PHALANX (com concorrência)



(a) Leitura para t = 1.

(b) Leitura para t = 2.

Figura 5.5: Desempenho da leitura em rede local: MINIMAL-CORRETO *X* PHALANX (com concorrência)

Os altos desvios médios coletados da execução do PHALANX com concorrência (figuras 5.4 e 5.5) são causados pela alta contenção de processador nos computadores do ambiente de testes. De fato, tanto o protocolo de leitura como o de escrita do PHALANX usam assinatura, o que dificulta a extração de um resultado com menor desvio médio em situação de concorrência em um ambiente com insuficiência de computadores para abrigar um número adequado de processos por máquina.

5.3.3 Algoritmos que tratam clientes bizantinos

Se o desenvolvimento de registradores com fortes semânticas de consistência já é uma tarefa complicada quando o contexto de faltas do sistema assume apenas servidores faltosos, esta situação fica mais complicada quando se assumem também clientes maliciosos. Alguns protocolos de BQS implementam registradores atômicos levando em conta a presença de clientes bizantinos, que tentam de algum modo atrapalhar o funcionamento dos algoritmos, seja em suas propriedades de consistência do sistema (*safety*) ou de terminação (*liveness*). Uma abordagem comum destes protocolos é usar mensagens assinadas a fim de que possam detectar modificações realizadas por clientes bizantinos.

Adicionalmente, de acordo com cada protocolo, outros mecanismos podem ser usados.

Em [34], é apresentado uma versão estendida do protocolo SBQ-L — chamado aqui de MINIMAL-FALTOSO — como o primeiro protocolo de BQS que a implementar registradores com fortes semânticas de consistência e tolerar alguns cenários de falta em clientes. Conforme visto na seção 5.3.2, o algoritmo SBQ-L usa o padrão de comunicação *listener* e quóruns assimétricos para realizar suas operações de leitura e escrita.

O protocolo MINIMAL-FALTOSO supõe clientes que possam explorar maliciosamente o padrão *listener*: escritores faltosos que podem tentar escrever diferentes valores, evitando que os servidores retornem o mesmo valor e, conseqüentemente, impedindo que leituras concorrentes ou futuras consigam terminar (**escrita venenosa**). Para confrontar tais clientes maliciosos, o MINIMAL-FALTOSO emprega mensagens assinadas e eco de mensagens entre servidores no seu protocolo de escrita. Nessa abordagem, mesmo com o risco maior de conluio entre clientes faltosos (ver comentário na seção 3.6.2.1), clientes compartilham a mesma chave privada e servidores possuem a chave pública correspondente, com a qual podem verificar e somente aceitar requisições de escrita que estejam corretamente assinadas. Além disso, para que se mantenha a consistência do sistema, os servidores realizam um novo passo em seu protocolo, transmitindo valores atualizados para os demais servidores.

Em um trabalho mais atual [27], é descrito o algoritmo BFT-BC, que viabiliza a implementação de um registrador atômico com semântica de acesso "vários usuários" e que trata uma grande variedade de problemas causados por clientes bizantinos. Este algoritmo pressupõe um sistema de quórum de f-disseminação, que emprega $n \geq 3f+1$ servidores no sistema e $2f+1$ servidores em quóruns.

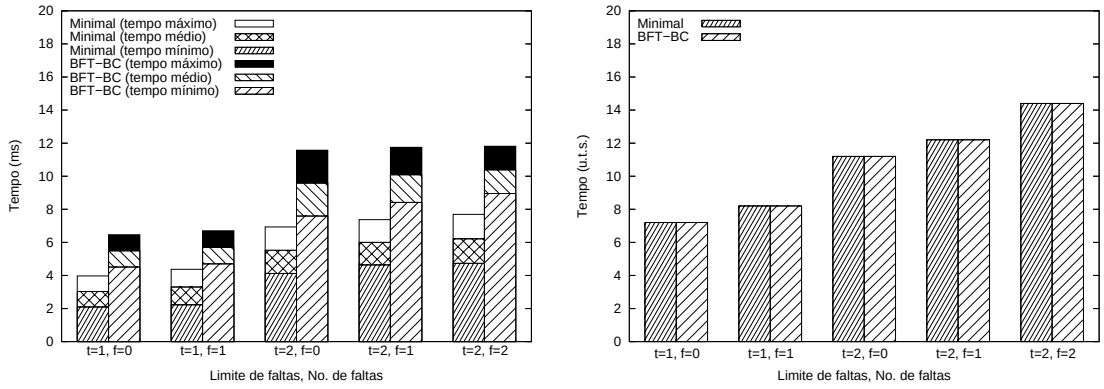
A fim de lidar com clientes bizantinos e preservar as suas fortes semânticas de consistência, o BFT-BC utiliza um mecanismo de provas assinadas em todas as suas etapas de execução. Desta maneira, para o cliente ingressar em uma nova fase do algoritmo, é preciso que ele apresente uma prova de que completou a fase anterior. Esta prova nada mais é que o conjunto de mensagens de resposta assinadas, coletadas de um quórum de servidores na fase anterior. Por exemplo, para o cliente escrever no quórum, é preciso que ele tenha terminado uma escrita anterior.

Usando esta técnica de provas, o BFT-BC emprega uma leitura otimizada com 2 passos de comunicação (contra 3 do MINIMAL-FALTOSO). Nesta situação, o cliente consegue retornar de um quórum um conjunto válido de mensagens com o mesmo par $\langle v,t \rangle$. Caso os pares retornados não sejam os mesmos, a leitura do BFT-BC requer 4 passos de comunicação, exigindo os passos adicionais de reescrita e espera por confirmações de um quórum. A escrita do BFT-BC processa em 6 passos em um cenário normal, onde se realizam as fases de consulta, preparação de escrita, escrita propriamente dita e espera por confirmações dos servidores. Em um protocolo otimizado de escrita, são necessários apenas 4 passos de comunicação, onde o cliente executa em um único acesso as etapas de consulta ao quórum e preparação da escrita. No MINIMAL-FALTOSO, é preciso 4 passos de comunicação para concluir o procedimento de escrita.

O foco deste experimento é observar os efeitos no desempenho dos protocolos MINIMAL-FALTOSO e BFT-BC a partir da maneira como ambos lidam com clientes bizantinos para preservar suas propriedades de corretude e atomicidade: o primeiro protocolo, mantendo consistência entre servi-

dores usando eco de mensagens entre servidores e um esquema de assinatura na escrita; o segundo, um conjunto de provas para atestar cada operação realizada pelos clientes. Este experimento aponta ainda o número de mensagens adicionais geradas por clientes corretos em suas operações de leitura quando da presença de concorrência no sistema.

A figura 5.6 ilustra os resultados da execução de operações de leitura sem concorrência. Nas execuções em rede local (figura 5.6(a)), sob todas as condições de falta no sistema, o algoritmo MINIMAL-FALTOSO obteve um desempenho melhor do que o algoritmo BFT-BC. Tal fato é um efeito da verificação criptográfica no BFT-BC de cada valor consultado do quórum, o que não acontece no algoritmo MINIMAL-FALTOSO, que, tal como o MINIMAL-CORRETO, não utiliza métodos adicionais em seu protocolo de leitura para tratar clientes bizantinos. Na presença de servidores faltosos, ambos os protocolos sofrem pequenas alterações em seus tempos de execução em relação a um cenário sem faltas, conseqüência do processamento adicional de mais uma mensagem, já que os valores recebidos dos servidores faltosos são descartados. Considerando um modelo de rede de larga escala (figura 5.6(b)), verifica-se que os tempos de latência nos protocolos de leitura tendem a ser iguais, pois os custos de processamento local tornam-se mínimos, e o número de passos de comunicação considerados para o término das leituras é sempre o mesmo. Ou seja, 2 passos para o BFT-BC (não há reescrita) e para o MINIMAL-FALTOSO (o cliente não espera por confirmações de um quórum quando de sua notificação aos servidores para cancelar o seu registro nos conjuntos de *listeners*).



(a) Leitura em rede local.

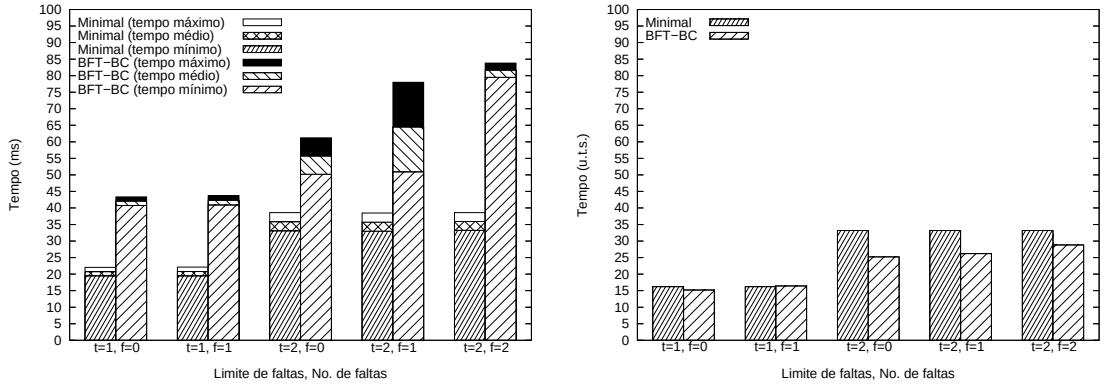
(b) Leitura em rede simulada, $\lambda = 0.1$.

Figura 5.6: Desempenho da leitura: MINIMAL-FALTOSO *X* BFT-BC (sem concorrência)

Para o protocolo de escrita (figura 5.7), observamos um melhor desempenho do MINIMAL-FALTOSO em todos os casos de falta. Sem concorrência, a escrita do BFT-BC acontece sempre pelo protocolo otimizado, que se realiza em 4 passos de comunicação, mesmo número usado pelo escritor no MINIMAL-FALTOSO. Portanto, a diferença expressa nos resultados adveio do uso de mecanismos adicionais nos dois protocolos avaliados, onde o efeito do uso de provas assinadas no BFT-BC consegue ser mais acentuado do que a assinatura nos clientes e a troca de mensagens entre servidores no MINIMAL-FALTOSO, principalmente quando o limite de faltas do sistema é igual a 2.

De fato, em uma execução em rede local, perde-se mais tempo em processamento local no BFT-BC por causa da quantidade de assinaturas realizadas durante a execução do seu protocolo. Neste algoritmo, cada servidor assina duas vezes: uma na resposta à mensagem de preparação da escrita do cliente, outra na confirmação da escrita deste. No algoritmo MINIMAL-FALTOSO, assina-se apenas uma vez quando o cliente realiza a sua requisição de escrita. Durante o processamento da requisição de escrita, embora cada servidor no MINIMAL-FALTOSO verifique e ecoe cada mensagem recebida (seja do cliente ou de outros servidores), este impacto é muito pequeno por dois motivos. Em primeiro lugar, porque sabidamente o custo com a transmissão de mensagens é baixo; em segundo lugar, porque o custo com verificação em nosso ambiente de execução é bem menor do que o custo com assinatura ($\approx 0,9$ ms contra ≈ 14 ms).

Em uma rede de larga escala, esta situação se altera. A figura 5.7(b) apresenta os resultados da escrita dos dois protocolos, onde o MINIMAL-FALTOSO, geralmente, obteve tempos de latência maior do que BFT-BC. Isto se justifica pelo alto custo da troca de mensagens entre servidores no MINIMAL-FALTOSO em um cenário onde a transmissão de mensagens é mais dispendiosa do que o tempo de execução local dos processos. Por este motivo também, o efeito de maior contenção de processamento provocado pelo maior uso de assinatura no BFT-BC em relação ao MINIMAL-FALTOSO tende a desaparecer, conferindo-lhe ao final melhor desempenho.



(a) Escrita em rede local.

(b) Escrita em rede simulada, $\lambda = 0.1$.

Figura 5.7: Desempenho da escrita: MINIMAL-FALTOSO *X* BFT-BC (sem concorrência)

A figura 5.8 apresenta os tempos coletados da execução de operações de escrita com leitores concorrentes em uma LAN. Em geral, o desempenho da escrita do MINIMAL-FALTOSO é melhor do que a escrita do BFT-BC. Levando em conta os desvios médios de cada execução, observam-se também alterações maiores no desempenho do escritor no BFT-BC à proporção que aumentam os leitores concorrentes. Em todas as condições testadas, os tempos (em valor aproximado) no MINIMAL-FALTOSO e no BFT-BC assumem, respectivamente, os seguintes intervalos: de 21 ms a 29 ms e 42 ms a 53 ms ($t = 1$); de 36 ms a 54 ms e de 56 ms a 102 ms (para $t = 2$).

O maior aumento na latência de escrita do BFT-BC quando do crescimento das cargas de concorrência e de faltas no sistema é um indício da maior contenção de processamento dos servidores

pelo uso de criptografia assimétrica em seus protocolos de leitura e escrita. No BFT-BC, os servidores, ao mesmo tempo em que verificam o conjunto de provas associado à requisição de escrita do cliente (recurso também usado no MINIMAL-FALTOSO), empregam, conforme já comentado nesta seção, assinaturas na resposta da preparação de escrita e na mensagem de confirmação de escrita do cliente. Adicionalmente, ao contrário do MINIMAL-FALTOSO, os servidores no BFT-BC podem empregar mecanismos de verificação e assinatura criptográficas também durante a leitura, nos casos em que esta operação é concorrente com escrita. Se examinarmos os mesmos casos de concorrência do BFT-BC na visão de um leitor, teremos uma confirmação deste impacto negativo no desempenho do escritor, causado pelo aumento de contenção nos servidores.

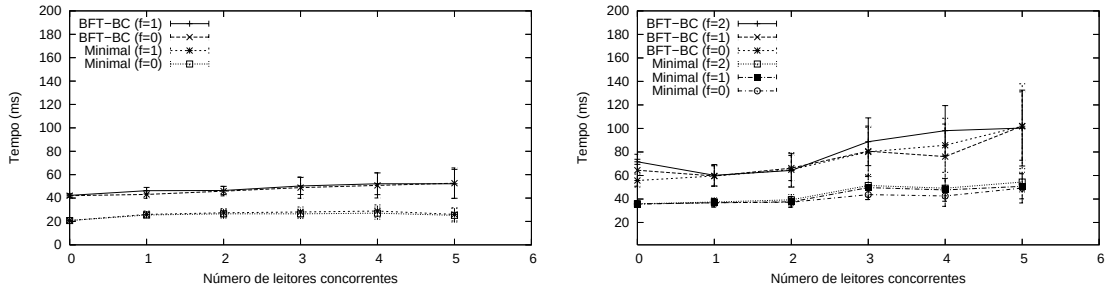
A figura 5.9 mostra resultados de experimentos em que um leitor concorre com um escritor e com um conjunto de 0 a 4 leitores. Percebe-se que, para todos os limites de faltas e até o número total de processos concorrentes experimentados, o desempenho da leitura é sempre pior no BFT-BC do que no MINIMAL-FALTOSO. Além disto, aquele algoritmo sempre mantém uma progressão maior em termos de latência do que este. Esta situação decorre da maior contenção de processamento nos servidores no BFT-BC durante a execução da leitura concorrente com as outras leituras e a escrita. Neste caso, cada servidor no BFT-BC realiza operações de assinatura na preparação da escrita e na confirmação da escrita do cliente concorrente, bem como na confirmação de cada reescrita dos leitores. As latências das operações de leitura possuem as seguintes variações: para $t = 1$, de $9,72 \pm 6$ ms a 29 ± 11 ms no BFT-BC contra $4 \pm 1,5$ ms a $16,3 \pm 5,7$ ms no MINIMAL-FALTOSO; para $t = 2$, de $27,4 \pm 11$ ms a $55 \pm 19,5$ ms no BFT-BC contra $10,4 \pm 6$ ms a $32,6 \pm 6$ ms no MINIMAL-FALTOSO.

O número de mensagens de reescrita geradas por este mesmo leitor no BFT-BC em geral manteve-se estável e muito baixo, o que indica um uso raro na prática. No MINIMAL-FALTOSO, assim como no MINIMAL-CORRETO (ver caso de análise da seção 5.3.2), a concorrência é notável em termos de mensagens adicionais geradas pelo mecanismo de *listener*, embora em número menor do que o MINIMAL-CORRETO (o protocolo de escrita correto possui um desempenho melhor do que a sua contraparte faltosa, o que provoca maior concorrência), mas sem afetar profundamente o desempenho total da leitura sob concorrência com escrita. A tabela 5.2 exibe o percentual de leituras realizadas com reescrita no BFT-BC e com o uso do padrão *listener* no MINIMAL-FALTOSO.

Protocolo / faltas	$t = 1$		$t = 2$		
	$f = 0$	$f = 1$	$f = 0$	$f = 1$	$f = 2$
MINIMAL-FALTOSO	64,2%	78,4%	82,5%	87,7%	91,7%
BFT-BC	6,2%	6,6%	11,8%	9,9%	9,5%

Tabela 5.2: Percentual de leituras com uso do padrão *listener* (MINIMAL-FALTOSO) e de reescritas (BFT-BC) – concorrência com 1 escritor e leitores

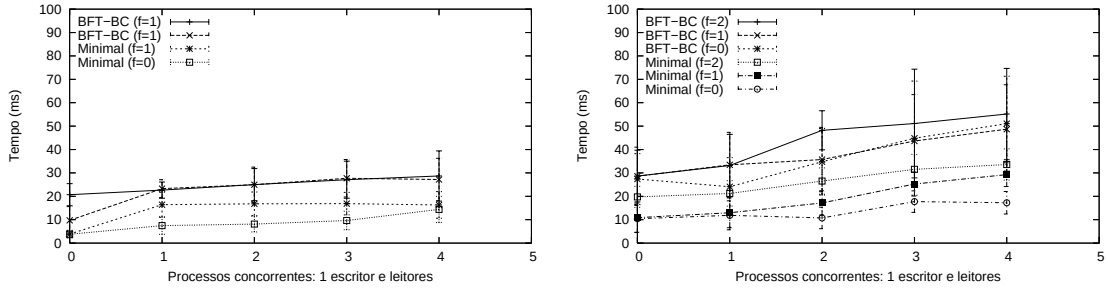
De maneira similar ao discutido na seção 5.3.2, aqui também ocorrem altos desvios médios nas latências do algoritmo BFT-BC em situação de concorrência. Isto se justifica pelo mesmo motivo apontado nos testes com PHALANX com concorrência, isto é, a grande contenção de processamento causada pelo uso de assinatura em um ambiente de rede local com insuficiência de máquinas para abrigar um número adequado de processos por máquina. O ideal certamente seria executar um processo do sistema em cada computador.



(a) Escrita para t = 1.

(b) Escrita para t = 2.

Figura 5.8: Desempenho da escrita em rede local: MINIMAL-FALTOSO *X* BFT-BC (com concorrência)



(a) Leitura para t = 1.

(b) Leitura para t = 2.

Figura 5.9: Desempenho da leitura em rede local: MINIMAL-FALTOSO *X* BFT-BC (com concorrência)

5.3.4 Analisando custo de armazenamento: BQS *X* Paxos

Existem duas técnicas que podem ser utilizadas para implementar replicação visando à tolerância a faltas bizantinas em sistemas de armazenamento: a *Replicação Máquina de Estados* (RME) [11, 41], introduzido na seção 2.3.1.1, e os *Sistemas de Quóruns* [32], introduzido na seção 2.3.1.2 e discorrido no capítulo 3. As diferenças entre estas duas técnicas podem ser resumidas em dois pontos: (a) replicação máquina de estados pode ser utilizada na implementação de qualquer serviço determinista, enquanto sistemas de quóruns podem implementar apenas armazenamento (operações de leitura e escrita); (b) replicação máquina de estados requer a resolução de consenso, o que exige algumas premissas do ambiente (ou protocolos com terminação probabilista) [17], enquanto sistemas de quóruns podem ser implementados em sistemas assíncronos. Estas diferenças têm fomentado um debate na comunidade de sistemas distribuídos a respeito da "ineficiência" do modelo de máquina de estados e da busca por alternativas a este modelo, dentre as quais se destacam os sistemas de quóruns bizantinos [1, 15].

Trabalhos recentes têm explicitado as vantagens e desvantagens destas técnicas quando compara-

das, exaltando tanto o caráter geral da máquina de estados [15] quanto a possibilidade de implementação dos sistemas de quóruns com quase nenhuma premissa [48] e sua potencial escalabilidade [1]. A literatura sobre a construção de sistemas tolerantes a faltas bizantinas tem apresentado alguns avanços interessantes no que tange a ambas as técnicas, dentre os quais podemos citar: a demonstração de que a replicação máquina de estados tolerante a faltas bizantinas pode ser implementada de forma eficiente [11, 36]; as novas otimizações descobertas para o protocolo de consenso PAXOS Bizantino [33, 49]; e os novos protocolos para sistemas de quóruns bizantinos que toleram clientes maliciosos utilizando um número ótimo de servidores [10, 27].

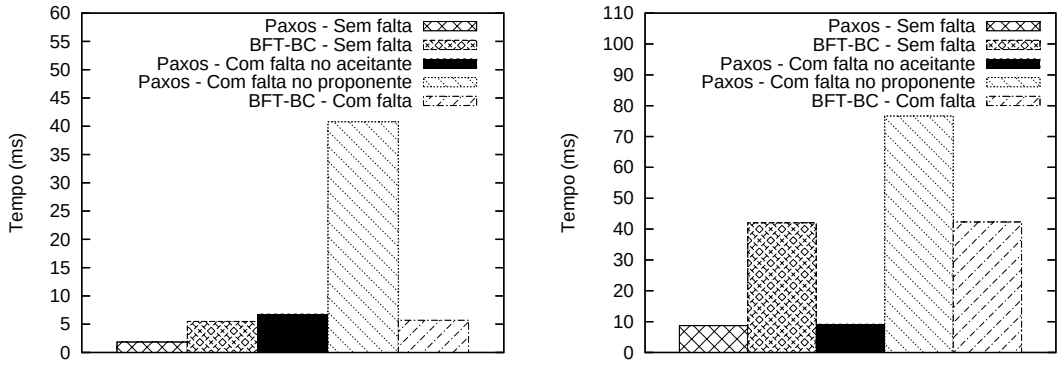
Estes avanços sugerem que as duas técnicas podem ser utilizadas na implementação de serviços confiáveis. Eles também instigam algumas perguntas: qual destas técnicas é a mais eficiente? Em que condições uma destas técnicas deve ser usada em detrimento a outra?

Neste caso de avaliação, investigamos esta questão através da avaliação experimental de dois dos protocolos mais eficientes e completos para concretização destas técnicas: PAXOS Bizantino [11, 25] (replicação máquina de estados) e BFT-BC [27], apresentado na seção 3.4.2.3 (sistemas de quóruns). O protocolo PAXOS Bizantino usado aqui agrega um conjunto de modificações para terminação rápida (em dois passos de comunicação) [33, 49]. Este protocolo foi escolhido devido ao seu bom desempenho em casos livres de falha e à sua resiliência ótima. Sua implementação foi realizada, assim como o BQSNEKO, usando o NEKO, aplicando funcionalidades similares às aplicadas naquele arcabouço, como o uso de uma camada de criptografia para execuções em rede real.

A figura 5.10 apresenta os resultados da leitura e escrita sem concorrência em ambiente de rede local e com diferentes condições de falta. O caso de leitura é apresentado pela figura 5.10(a). Em um cenário sem faltas, ambos os protocolos executam de forma otimizada, terminando em apenas dois passos. Neste caso, o PAXOS alcança um desempenho um pouco melhor do que o BFT-BC. Esta diferença de desempenho deve-se ao uso de verificação criptográfica no BFT-BC durante a consulta de dados dos servidores, o que não é feito no PAXOS, que, nesta condição, não utiliza operações criptográficas. No caso com falta, a latência de leitura do BFT-BC é praticamente indiferente a mudanças (de $\approx 5,5$ ms para $\approx 5,7$ ms), pois o cliente apenas contabiliza o tempo de espera por uma nova mensagem do quórum — que é baixo por se tratar de uma rede local — e o custo para verificação (menos de 1 ms) de mais um par $\langle v,t \rangle$ vindo de um servidor correto. Já no PAXOS, com falta no aceitante e efetuando o procedimento otimizado em 50% dos casos, em que o cliente realiza a sua leitura em apenas 2 passos de comunicação esperando $n-f$ mensagens com o mesmo valor, o cliente se vê obrigado a realizar a sua leitura por meio de uma requisição usando difusão com ordem total, acarretando um aumento de latência de aproximadamente $1,85$ ms (caso sem faltas) para $6,8$ ms. Quando este processo faltoso, além de aceitante, é um proponente do *round* recém-iniciado, o tempo de execução do PAXOS aumenta ainda mais: passa para cerca de 41 ms, pois o protocolo de difusão com ordem total será levado para um segundo *round*, o que envolve um procedimento extra para seleção de um novo proponente.

No caso da escrita sem concorrência (figura 5.10(b)), em um situação sem faltas, o BFT-BC tem uma latência bem maior do que o PAXOS. Isto se explica pelo alto custo das operações de assinatura

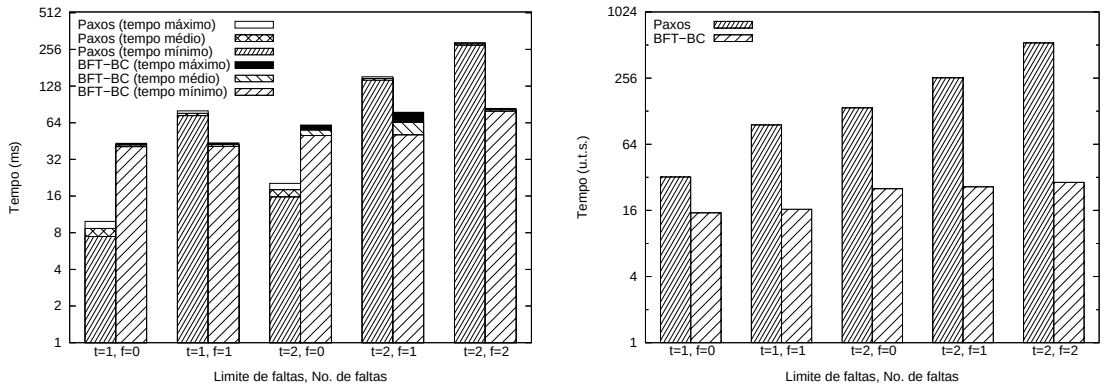
RSA usadas pelo BFT-BC para a preparação da escrita. No caso de faltas, um escritor no PAXOS não sofre maior pertubação quando confrontado com uma falta no aceitante (situação de *round* favorável). Ou seja, em uma rede local, na escrita, não há praticamente diferença entre um *round* muito favorável e um *round* favorável (um passo de comunicação a mais). A escrita do BFT-BC também não se altera pelo mesmo motivo apresentado para a sua leitura. O PAXOS somente se desempenha pior do que o BFT-BC na escrita, apenas quando ocorre uma falha no proponente. Nesta situação, tal como apresentado na leitura, um novo proponente será escolhido, cujo processo envolve, além de dois passos adicionais de comunicação, o uso de operação de assinatura *RSA*, reconhecidamente custosa em ambientes de rede local. Note que, no PAXOS, o valor da latência de leitura com falha no proponente (figura 5.10(a)) é praticamente a metade (≈ 41 ms) da latência de escrita com falha no proponente (≈ 77 ms). Isto acontece porque ocorreram leituras não otimizadas em 50% das vezes, situações em que o houve difusão com ordem total, oportunizando a atuação do proponente faltoso.



(a) Operação de leitura.

(b) Operação de escrita.

Figura 5.10: Desempenho dos protocolos de leitura e escrita em rede local (sem concorrência e $t = 1$): PAXOS *X* BFT-BC



(a) Escrita em rede local.

(b) Escrita em rede simulada, $\lambda = 0.1$.

Figura 5.11: Desempenho da escrita em rede local sem concorrência: PAXOS (com falta no proponente) *X* BFT-BC

Para mostrar o quão sensível a uma falta no proponente é o protocolo PAXOS, a figura 5.11 ilustra uma comparação entre os tempos das escritas do BFT-BC e do PAXOS em situações sem concorrência e com limites de faltas $t = 1$ e $t = 2$. A figura 5.11(a) mostra as execuções em rede local, e a figura 5.11(b), em rede de larga escala simulada. Fica bastante claro por estes gráficos que, para todas as condições de limite de faltas e ambiente de rede, a presença do proponente faltoso causa um aumento significativo no tempo de terminação da escrita do PAXOS. Vale observar que, no ambiente de larga escala, o PAXOS, além de sofrer um impacto maior com a falha no proponente, também apresenta as maiores latências porque executa com uma complexidade de mensagens em $O(n^2)$ (contra $O(n)$ do BFT-BC). A tabela 5.3 exibe somente os valores de latência em ambos os protocolos.

Protocolo / faltas	$t = 1$		$t = 2$		
	$f = 0$	$f = 1$	$f = 0$	$f = 1$	$f = 2$
PAXOS, LAN	$\approx 8,7$ ms	$\approx 76,6$ ms	≈ 18 ms	$\approx 147,8$ ms	≈ 284 ms
BFT-BC, LAN	≈ 42 ms	$\approx 42,3$ ms	$\approx 55,7$ ms	$\approx 64,5$ ms	$\approx 81,6$ ms
PAXOS, larga escala	32,2 u.t.s.	95,9 u.t.s.	137,2 u.t.s.	257,2 u.t.s.	534 u.t.s.
BFT-BC, larga escala	15,2 u.t.s.	16,4 u.t.s.	25,2 u.t.s.	26,2 u.t.s.	28,8 u.t.s.

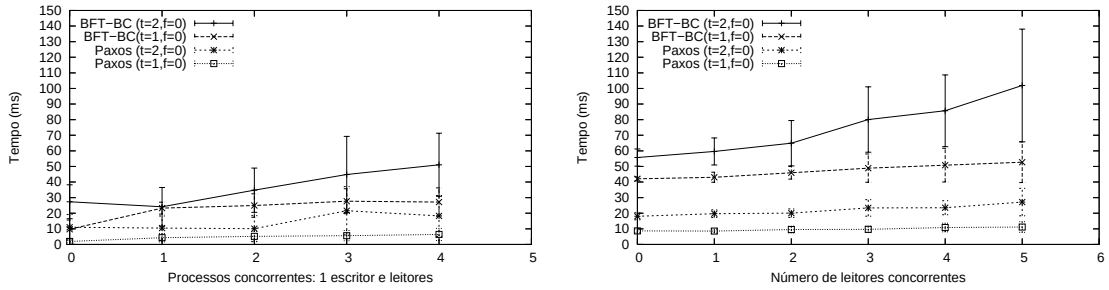
Tabela 5.3: Latências de escrita no PAXOS (com faltas no proponente) e no BFT-BC – redes local e larga escala simulada, sem concorrência.

A figura 5.12(a) exibe situações de concorrência realizadas em uma rede local e considerando apenas cenários com servidores corretos. A figura 5.12(a) apresenta os resultados das operações de um leitor concorrendo com um escritor e com 0 a 4 leitores, enquanto que a figura 5.12(b) mostra os resultados de um escritor concorrendo com 0 a 5 leitores. Em todos os casos, considerando os desvios médios calculados para cada cenário, o PAXOS obteve um melhor desempenho do que as suas contrapartes do BFT-BC. Isto sinaliza uma menor contenção de processamento e uma melhor escalabilidade do serviço de armazenamento usando a abordagem com replicação máquina de estados. O algoritmo representando a replicação máquina de estados teve as suas latências variando em média de $1,88$ ms a $6,38$ ms ($t = 1$) e 11 ms a 18 ms ($t = 2$) na leitura; de $8,67$ ms a $11,15$ ms ($t = 1$) e 18 ms a $27,16$ ms ($t = 2$) na escrita. O algoritmo representando o sistema de quóruns teve as latências variando de $9,72$ ms a $27,18$ ms ($t = 1$) e de $27,36$ ms a $51,06$ ms ($t = 2$) na leitura; de 42 ms a $52,7$ ms ($t = 1$) e de $55,7$ ms a $101,9$ ms na escrita.

Aponte-se ainda aqui que, nas operações de leitura da figura 5.12(a), em cerca de 88% das vezes foram feitas leituras otimizadas para um cenário com $t = 1$ (4 servidores) e, para $t = 2$ (7 servidores), este número de leituras otimizadas caiu para cerca de 64% das vezes. Estes dados mostram que, assim como mostrado em seções anteriores, nas execuções de alguns algoritmos de BQS que não empregam assinatura de mensagens em seu funcionamento (como no MINIMAL-CORRETO na seção 5.3.2 e no MINIMAL-FALTOSO na seção 5.3.3), as operações do cliente com o PAXOS conseguem apresentar pouca variação de desempenho em situação de concorrência, embora isto não signifique uma fraca concorrência em termos de mensagens adicionais geradas pelos seus protocolos correspondentes. Em particular, o PAXOS demonstrou escalabilidade nos testes realizados porque a implementação corrente não realiza um consenso para cada requisição recebida do cliente, mas para um um agrupamento de requisições ordenadas periodicamente e que são executadas em lote (*batch*), caso o conjunto de

requisições obtido não seja vazio.

Durante as execuções com processos concorrentes, ocorrem pertubações no desvio médio no BFT-BC conforme relatado na seção 5.3.3. Observa-se, mais uma vez, que isto acontece, pois, nos casos de concorrência, há maior contenção de processamento pelo uso de assinaturas nos seus protocolos, somada à insuficiência de um número adequado de máquinas no nosso ambiente de testes para abrigar todos os processos.



(a) Leitura com 1 escritor e leitores.

(b) Escritor com leitores.

Figura 5.12: Desempenho de leitura e escrita em rede local: PAXOS *X* BFT-BC (com concorrência e sem faltas)

5.4 Considerações finais

O presente capítulo exibiu e discutiu uma série de experimentos envolvendo algoritmos de sistemas de quóruns bizantinos usando o arcabouço BQSNEKO, legitimando-o a princípio como uma ferramenta capaz de ser explorada para avaliação de algoritmos de sistemas de quóruns bizantinos. Embora não existam trabalhos na literatura que realizem avaliações entre algoritmos de BQS, tal como o apresentado neste capítulo, nem ferramentas similares para realizar análises comparativas, cremos que o BQSNEKO consegue atender às expectativas que motivaram a sua construção por dois motivos básicos: *(i)* seu suporte de execução e desenvolvimento é o NEKO, cuja validação é demonstrada pela quantidade de trabalhos teóricos indiretamente relacionados ou experimentais que o utilizam diretamente como ferramenta (vide seção 4.5); *(ii)* os resultados apresentados neste capítulo, ainda que prejudicados em poucos momentos pela já admitida limitação de recursos de *hardware*, ponderados os pressupostos dos experimentos, quando não se apresentam certamente absurdos, não ferem em absoluto certos conceitos sedimentados no que concerne a algoritmos distribuídos. Um exemplo é o fato de que algoritmos $O(n)$ são mais eficientes do que algoritmos em $O(n^2)$. Neste caso, os resultados apenas relativizam estas noções a partir do contexto de estudo desta dissertação (sistema de armazenamento bizantino, especificamente sistema de quóruns bizantinos) e das premissas que serviram de base aos experimentos (e.g., baixo número de réplicas) mostrando situações que normalmente não são consideradas no caso geral e que são importantes no presente trabalho, como a importância de se contabilizar o processamento local quando do uso de assinaturas criptográficas em certos algoritmos de BQS.

Ao mesmo tempo, este capítulo representa um passo salutar em direção a uma melhor compreensão do funcionamento das técnicas correntes aplicadas a sistemas de armazenamento tolerantes a faltas bizantinas, sobretudo de sistemas que empregam a abordagem de sistemas de quóruns bizantinos. Neste aspecto, com base nos resultados obtidos neste capítulo, alguns pontos a respeito dos mecanismos usados nos protocolos experimentados (e, em última análise, dos próprios protocolos) podem ser salientados.

Primeiramente, há de se ressaltar o grande impacto causado pelo uso de assinaturas criptográficas em protocolos para armazenamento de dados tolerantes a faltas bizantinas quando se trata de uma rede local, onde o gargalo de desempenho se localiza no processamento local e a tempo para comunicação é baixo. Normalmente, considera-se o custo com computação local desprezível, porém, em certos algoritmos de BQS em que o uso de assinatura é um fato, tal dispêndio com computação tem que ser considerado. Esta constatação penaliza a priori os protocolos que armazenam dados auto-verificáveis, como o PHALANX e o BFT-BC, o que pôde ser percebido, para o PHALANX, em sua comparação com o MINIMAL-CORRETO (seção 5.3.2); e, para o BFT-BC, em sua comparação com o MINIMAL-FALTOSO (seção 5.3.3) e com o PAXOS (seção 5.3.4). Por outro lado, percebeu-se também que este custo com assinatura pode ser atenuado quando estes mesmos algoritmos são colocados em um ambiente de rede de larga escala.

No primeiro caso, embora o MINIMAL-CORRETO e o PHALANX detenham propriedades teóricas iguais no que concerne ao número de passos (ambos executam sua escrita em 4 passos e sua leitura sem concorrência efetivamente em 2 passos) e na complexidade de mensagens ($O(n)$ para ambos), o primeiro algoritmo mostrou-se mais eficiente do que o segundo em todas as configurações de execução feitas em rede local. Isto ocorreu justamente porque MINIMAL-CORRETO não emprega operações criptográficas em seus procedimentos de leitura e escrita, sendo que esta diferença ao seu favor ficou mais notável à medida que se imputava uma maior carga de concorrência e de faltas no sistema. No segundo caso, o uso de assinaturas no BFT-BC acabou por prejudicar o seu desempenho, principalmente na escrita e em situação de concorrência. Em comparação ao MINIMAL-FALTOSO, o BFT-BC teve o seu desempenho na escrita mais afetado quando do aumento de processos concorrentes, fazendo com que a sua latência correspondente, que já era maior em situações sem concorrência, esboçasse uma diferença maior em seu desfavor. Em comparação ao PAXOS, o BFT-BC, notadamente nas operações de escrita, apresentou maiores latências. Sua situação ficou mais crítica quando da adição de processos concorrentes, o que, mais uma vez demonstra o quão oneroso é empregar assinatura nos protocolos dentro de um ambiente de rede local.

Demonstrou-se a eficiência dos protocolos que implementam registradores “mínimos”, ou seja, os algoritmos MINIMAL-CORRETO e MINIMAL-FALTOSO, que usam o padrão *listener* na leitura para alcançar atomicidade de operações. Estes algoritmos alcançaram uma eficiência maior em relação às suas contrapartes experimentadas, mesmo quando, em contextos de concorrência, o número de mensagens adicionais trocadas na rede local aumentou. Desta forma, aproveitando o inerente custo baixo no transporte de mensagens em uma rede local, pode-se concluir que o uso do mecanismo de *listener* e quóruns assimétricos é mais eficiente do que o uso da reescrita e quóruns simétricos. É bastante provável que esta superioridade do *listener* não vigore em uma rede de larga escala (como já é vaticinado nos testes com operações de leitura em rede de larga escala simulada na seção 5.3.2),

visto que, neste âmbito, o custo de processamento (mesmo com as assinaturas) tende a ser muito menor que o custo de transporte de mensagens.

Adicionalmente, mostrou-se que protocolos que executam com complexidade $O(n^2)$ não são necessariamente mais custosos. Como já apontado, em uma rede local e com um número pequeno de réplicas, foi visto que a latência de escrita do algoritmo SWMR-SEGURO usando difusão entre servidores foi menor do que o algoritmo MWMR-SEGURO, que executa em $O(n)$, mas utiliza um esquema de assinatura de mensagens sabidamente oneroso em redes locais. No caso comum, quando se considera desprezível o custo com processamento local (ou no caso de um grande número de réplicas), algoritmos em $O(n^2)$ certamente possuem um desempenho pior do que algoritmos com complexidade de mensagens $O(n)$.

Por fim, refletiu-se a eficiência da abordagem de implementação de armazenamento bizantino usando replicação máquinas de estado pelo protocolo PAXOS à proporção que suas execuções ocorrem em rede local e não ocorrem faltas no processo proponente. Esta noção diverge das opiniões de que a abordagem com sistemas de quóruns são sempre mais eficientes do que a replicação máquinas de estados. De fato, as operações do PAXOS, que, normalmente, ocorrem com complexidade de $O(n^2)$, em casos de *rounds* muito favoráveis, demonstraram possuir uma boa escalabilidade, factível por conta da implementação em lotes. Resta saber se estas boas eficiência e escalabilidade ficarão asseguradas em um ambiente de larga escala e em situações com falta.

Capítulo 6

Conclusão

O presente capítulo conclui esta dissertação começando com a revisão dos seus objetivos, que foram citados no capítulo introdutório, apresentando as reflexões terminais a respeito do desenvolvimento deste trabalho e encerrando com a exposição de alguns trabalhos futuros.

6.1 Revisão dos objetivos e comentários finais

Este trabalho apresentou um arcabouço de avaliação de algoritmos de sistemas de quóruns bizantinos, denominado BQSNEKO. O capítulo 4 descreve o que se coloca como o objetivo principal desta dissertação: a implementação do arcabouço de avaliação de algoritmos de BQS, denominado BQSNEKO. Podemos verificar inicialmente em tal capítulo uma breve descrição do suporte de desenvolvimento e execução do BQSNEKO, o *framework* NEKO. Em seguida, encontramos um detalhamento da arquitetura do arcabouço BQSNEKO, que explora a infra-estrutura do NEKO provendo facilidades para implementação de algoritmos de BQS e para construção de perfis de falta bizantina. Estas novas funcionalidades possibilitam a execução e a posterior avaliação destes algoritmos de acordo com variados contextos de execução. Os contextos de execução podem variar, por exemplo, por suas características de rede — abrangendo modelos de redes reais ou simuladas já implementadas pelo (acrescentadas ao) próprio NEKO — ou por sua carga de falhas. Convém enfatizar que estas facilidades para criação de algoritmos de BQS e de perfis de falta bizantina, conforme discutido no citado capítulo, inexistem no próprio NEKO. Ademais, na própria literatura, apesar da grande quantidade de abordagens de protocolos de BQS (por exemplo, [10, 27, 32, 34]), é possível perceber tanto a ausência de trabalhos que comparem tais protocolos quanto a ausência de ferramentas que sirvam para tal fim de análise, o que evidencia a contribuição do presente trabalho. Ao final do capítulo 4, ainda é apresentado um exemplo de como se implementar um novo protocolo de BQS e um novo perfil de falta bizantina usando o BQSNEKO, além de como configurar e executar uma instância deste protocolo recém-implementado injetando faltas bizantinas em seu contexto de execução.

O capítulo 5 contempla o que se considerou na introdução desta dissertação como o seu primeiro objetivo específico: a avaliação de protocolos para BQS por meio da apresentação de resultados de

experimentos divididos em 4 casos de análise. Em um primeiro momento, isto atesta a utilidade desta solução para avaliação de algoritmos de BQS, atingindo retroativamente o objetivo primário deste trabalho, de apresentar uma ferramenta capaz de auxiliar o processo de desenvolvimento de sistemas de armazenamento usando BQS em sua fase inicial de construção, ou seja, durante as fases de projeto de tal sistema, por via de prototipação.

Em um segundo momento, o capítulo 5 contribui, em uma última análise, no entendimento de abordagens empregadas para implementação de sistemas de armazenamento tolerantes a faltas bizantinas. Diga-se que esta contribuição é muito importante, sobretudo para a classe dos sistemas de quóruns bizantinos (foco central desta dissertação). Sabendo-se que existe um grande número de trabalhos que apresentam protocolos de sistemas de quóruns bizantinos, é preciso que a sua efetividade e, fundamentalmente, seu comportamento sejam observados na prática, tendo em vista as possibilidades oferecidas por diferentes cenários de execução — diferentes modelos de rede, diferentes alterações na carga de faltas, presença de concorrência de operações, etc. E, para que esta atividade de avaliação aconteça em um caráter mais preciso e de maneira mais eficiente, convém realizá-lo em um mesmo ambiente: isto evita maiores esforços com desenvolvimento e maiores problemas com comparações entre implementações realizadas sobre suportes computacionais (ora em *software*, ora em *hardware*) diferentes e específicos, onde se tem o risco maior de comprometer a qualidade da análise. O trabalho em [20], por exemplo, ilustra este último aspecto. Tal compara duas implementações de sistemas de armazenamento tolerantes a faltas bizantinas: uma baseada em sistema de quóruns (apresentada no mesmo trabalho), outra baseada em replicação máquinas de estado realizada por outro trabalho [9], porém se trata de análise de sistemas específicos e diferentes.

Finalmente, o capítulo 3 contempla o que se definiu como segundo objetivo específico na introdução desta dissertação. Neste, é possível achar um "*survey*", que busca organizar o conhecimento até então sobre sistemas de quóruns bizantinos utilizando um formato descritivo próprio. Trata-se de um documento dentro da dissertação a respeito dos principais protocolos de BQS já propostos, suas notações algorítmicas e as suas respectivas propriedades teóricas.

Em um ponto de vista teórico, o capítulo 3 auxilia ainda na compreensão dos algoritmos de BQS e de suas propriedades sob uma perspectiva modular. Isto é, a partir do momento em que se direciona a observação ao objeto de estudo (ou seja, os Sistemas de Quóruns Bizantinos e os seus algoritmos) dentro de uma evolução, que coincide aqui com a evolução cronológica dos trabalhos descritos, consegue-se refletir sobre o que cada abordagem acrescentou no conhecimento da área.

Desta maneira, temos a possibilidade de pensar nos algoritmos de BQS como uma estrutura genérica (tal como a descrita na seção 3.3), que, juntamente com um contexto de execução (e.g., um contexto de falhas específico) poderia ser "preenchida" por módulos ou componentes básicos correspondentes a propriedades específicas: uma vez compostos, este conjunto de módulos seriam capazes de oferecer determinadas propriedades à estrutura resultante, o algoritmo construído. Por exemplo, se pensarmos na conjunção de módulos com semânticas "escrever dado auto-verificável" e "consultar dado auto-verificável" dentro de um contexto com apenas clientes corretos, poderíamos construir um algoritmo com semântica regular, já que todo cliente, que seria correto, leria, em caso trivial, o último realmente escrito no sistema e, em caso de concorrência, um dos dados sendo escritos. A esta

primeira versão de algoritmo, se acrescentássemos um terceiro módulo de nome “reescrever dado auto-verificável”, ofereceríamos um semântica atômica: agora, na concorrência, todos os clientes até a próxima escrita leriam o último dado escrito no sistema.

Em suma, espera-se que esta dissertação possa colaborar de alguma forma com os seguintes tópicos: com o capítulo 3, no maior conhecimento da abordagem de Sistemas de Quóruns Bizantinos; com o capítulo 4, na disponibilização de uma ferramenta útil de avaliação de algoritmos de BQS; com o capítulo 5, no fortalecimento da discussão de resultados dos experimentos envolvendo os algoritmos de BQS e, em um âmbito mais amplo, de mecanismos de tolerância a faltas bizantinas, sobretudo em sistemas de armazenamento.

6.2 Trabalhos futuros

Esta dissertação certamente não finaliza todos os pontos a que se propôs apresentar e discutir. Os tópicos a seguir apresentam alguns desdobramentos do trabalho atual:

1. **Aspectos de implementação**: existem muitos tópicos ainda pendentes no desenvolvimento do BQSNEKO. Além das atividades triviais de organização de código, outros melhoramentos são previstos, como: possibilidade de configuração de execuções com depuração de mensagens em arquivo de *log* (para realizar análises posteriores usando o número de mensagens extras de um certo tipo como métrica), sem necessitar de reprogramações intrusivas; criação de uma nova versão do BQSNEKO, hoje baseada na versão 0.9 do NEKO que utiliza um modelo de construção dos processos com camadas, usando a versão corrente do NEKO (versão 1.0 [1]), que utiliza um modelo de componentes distinto para instanciar processos e protocolos distribuídos; permitir em arquivo de configuração do BQSNEKO a escolha do número de bits da chave de criptografia assimétrica (hoje, o valor fixo é de 1024 bits). Como conseqüência das reflexões teóricas do capítulo 3, poderíamos reestruturar o arcabouço BQSNEKO de forma a ter e possibilitar efetivamente a implementação de módulos que, juntos e ao lado de um contexto pré-determinado de execução, conseguiriam construir algoritmos de BQS com certas propriedades teóricas. Estes componentes primários para implementação de protocolos de BQS ofereceriam determinadas funcionalidades, como, por exemplo, “consultar dados de um quórum”.

2. **Aspectos de avaliação de protocolos**: no que diz respeito aos testes com algoritmos de BQS mostrados no capítulo 5, ficam como trabalhos futuros a investigação mais apurada em ambientes de larga escala. Uma possível via de metodologia para experimentações de protocolos em ambientes de larga escala pode ser encontrada em [3]. Neste modelo de rede, onde o custo com processamento é baixo e o custo com transporte de mensagens é relevante, espera-se um melhor desempenho dos protocolos de quóruns (mesmo os que usam assinatura), que, normalmente, trocam menos mensagens.

   Outro tópico pendente no quesito de avaliação de algoritmos é a verificação do comportamento dos protocolos que utilizam assinatura mediante a mudança do algoritmo criptográfico e do

[1] Disponível em `http://ddsg.jaist.ac.jp/neko/`

tamanho da chave criptográfica. Em alguns casos, como em redes locais, teoricamente mais seguras, é possível usar chaves de criptografia com um número menor de bits (nos testes feitos, utiliza-se 1024 bits de chave), o que possivelmente permitiria uma otimização do resultado de algoritmos que empregam assinaturas como o BFT-BC e o PHALANX. Ainda neste aspecto, outro desdobramento possível deste trabalho é a avaliação de algoritmos alternativos ao BFT-BC, tais como os apresentados em [6, 10], que oferecem as mesmas garantias deste a custo de mais servidores e mensagens.

Apêndice A

Exemplo de código-fonte no BQSNEKO – Protocolo de leitura do cliente

Este apêndice apresenta o código-fonte no BQSNEKO do algoritmo de leitura do cliente (método *read*, ver página seguinte) do exemplo de implementação exibido na seção 4.4. O código-fonte corresponde ao protocolo mostrado na seção 3.4.2.2 e está divido em duas fases.

A fase 1 do algoritmo (linhas 2 a 28) comprende o envio de uma mensagem MRQueryMessage e a recepção de mensagens MRQueryRespSignedMessage de um quórum que são armazenadas na variável *list*. A seguir, o cliente coleta $f+1$ pares idênticos $\langle v,t \rangle$ de *list* com o maior *timestamp*. O par $\langle v,t \rangle$ propriamente é devolvido na variável *resp*. Se não for encontrado o par $\langle v,t \rangle$ desejado, o método retorna nulo. Se for encontrado algum par $\langle v,t \rangle$ (variável *resp* não nula), o cliente passa para a fase 2 (linhas 30 a 36). Na fase 2, este envia uma mensagem MRWriteBackListMessage a fim de iniciar o processo de reescrita em quórum. Note que o método *setSignature(boolean)* indica à camada de criptografia para não aplicar o custo de criptografia (isto é, cliente não assina). Por fim, o cliente espera por confirmações (mensagens MRUpdateAckMessage) de um quórum (não mostrado no código) .

Quadro A.1: Protocolo do cliente (algoritmo de leitura)

```
1  public Object read(){
2     do{
3        currentElm = (ReadWriteObject)list.get(base);
4        foundList = new ArrayList(0);
5        foundList.add(currentElm);
6        foundValue = foundDiff = false;
7        for(int k = 0; k < list.size(); k++){
8           if(k != base){
9              listElm = (ReadWriteObject)list.get(k);
10             if(listElm.equals(currentElm)){
11                foundList.add(listElm);
12             }
13             else{
14                if(!foundDiff && k > base){
15                   newBase = k;
16                   foundDiff = true;
17                }
18             }
19          }
20          /* Verificando se foundList tem pelo menos f+1 elementos */
21          if(foundList.size() >= (serversFaultsNum+1)){
22             resp = currentElm;
23             foundValue = true;
24             break;
25          }
26       }
27       base = newBase;
28    }while(!foundValue);
29
30    if(resp != null){
31       MRWriteBackListMessage newWriteBack;
32       newWriteBack = new MRWriteBackListMessage(foundList,resp,getCurrentNonce());
33       newWriteBack.setSignature(false);
34       send(new NekoMessage(myProcessId,qInfo.getServers(),
35                            newWriteBack, newWriteBack.getType()));
36    }
37    return resp;
38 }
```

Apêndice B

Exemplo de código-fonte no BQSNEKO – Protocolo de escrita do cliente

Este apêndice mostra o código-fonte no BQSNEKO do algoritmo de escrita do cliente (método *write*, ver página seguinte). A implementação está ligada ao exemplo mostrado na seção 4.4, que corresponde ao protocolo especificado na seção 3.4.2.2, sendo dividido em 3 fases.

Na fase 1 (linhas 2 a 25), o cliente envia uma mensagem MRQueryMessage e depois espera por mensagens MRQueryRespMessage na variável *list*. Na fase 2 (linhas 27 a 32), o cliente calcula o próximo *timestamp* a partir do par com o maior *timestamp* obtido anteriormente em *list*. A seguir, o cliente envia uma mensagem MRUpdateMessage. O argumento de MRUpdateMessage igual a *false* faz com que, na escrita do *log*, esteja indicado que a mensagem MRUpdateMessage é uma mensagem de escrita, não de reescrita. O cliente ainda espera um conjunto de ecos assinados de um quórum (mensagens MRUpdateEchoSignedMessage), que são armazenadas na variável *list*. Esta parte do código (não mostrado) é similar às linhas 2 a 25.

Na fase 3 (linhas 34 a 38), é enviada uma mensagem MRUpdateListMessage, contendo o conjunto recebido de ecos assinados. A chamada *setSignature(false)* indica à camada de criptografia que desconsidere o custo de assinatura, ou seja, o cliente não assina ao enviar. Depois, são esperadas confirmações de um quórum (mensagens MRUpdateAckMessage), que também não são mostradas em código, cujo passo também similar às linhas 2 a 25.

Quadro B.1: Protocolo do cliente (algoritmo de escrita)

```
1     public void write(Object v){
2      MRQueryMessage newQueryMsg = new MRQueryMessage(false,getNextNonce());
3       send(new NekoMessage(myProcessId,qInfo.getServers(),
4                            newQueryMsg, newQueryMsg.getType()));
5      do{
6          NekoMessage theMessage = receive("MRQueryRespMessage");
7          queryRespMsg = (MRQueryRespMessage)theMessage.getContent();
8          if(queryRespMsg.getNonce() == getCurrentNonce()){
9             Integer source = new Integer(theMessage.getSource());
10            if(sources.indexOf(source) == -1){
11               sources.add(source);
12               if(list.size() == 0)
13                  largestQueryResp = queryRespMsg;
14               else{
15                  queryRespObj = queryRespMsg.getObject();
16                  largestQueryRespObj = largestQueryResp.getObject();
17
18                  if(queryRespObj.compareTSTo(largestQueryRespObj) > 0)
19                     largestQueryResp = queryRespMsg;
20               }
21               list.add(queryRespObj);
22               i++;
23            }
24         }
25      }while(i < numServers && list.size() < rQuorumSize);
26
27      long ts = nextTimestamp(largestQueryResp.getObject().getTimestamp());
28
29      ReadWriteObject oNew = new ReadWriteObject(ts, (Integer)v);
30      MRUpdateMessage newUpdateMsg = new MRUpdateMessage(oNew,false,getCurrentNonce());
31      send(new NekoMessage(myProcessId,qInfo.getServers(),
32                           newUpdateMsg,newUpdateMsg.getType()));
33
34      MRUpdateListMessage = new MRUpdateListMessage(list,oNew,getCurrentNonce());
35      newUpdateListMsg.setSignature(false);
36      send(new NekoMessage(myProcessId,qInfo.getServers(),
37                           newUpdateListMsg,newUpdateListMsg.getType()));
38
39    }
40
41  }
```

Apêndice C

Exemplo de código-fonte no BQSNEKO – Protocolo do servidor

O presente apêndice exibe o código-fonte no BQSNEKO do servidor (método *execute*, dividido em parte 1 no quadro C.1 e parte 2 no quadro C.2, ver páginas seguintes). Esta implementação corresponde ao exemplo mostrado no seção 4.4, concretizando o protocolo do servidor especificado na seção 3.4.2.2.

A ação do servidor organiza-se de acordo com a mensagem recebida do cliente. O quadro C.1 apresenta as instruções a serem seguidas pelo servidor se este receber as mensagens MRQueryMessage e MRUpdateMessage, respectivamente nas requisições de consulta e de escrita do cliente. Perceba que, na linha 9, o servidor, ao responder uma requisição de consulta do cliente, não verifica a mensagem.

Já o quadro C.2 apresenta as ações do servidor quando este recebe as requisições de escrita pronta e reescrita do cliente. Note que, na recepção da requisição de escrita pronta, ou seja, da mensagem MRUpdateListMessage, o servidor verifica uma lista com um quórum de ecos assinados (recolhidos do passo anterior de escrita do cliente) usando o seu método *checkList(AbstractChallengeMessage)* (linha 4), cujo código não é exibido. Esta lista de ecos assinados é justamente a variável *list* contida no construtor da mensagem MRUpdateListMessage na linha 34 do algoritmo de escrita do cliente (quadro B.1). Da mesma maneira, quando recebe uma mensagem de reescrita, ou seja, uma mensagem MRWriteBackListMessage, o servidor verifica uma lista com $f + 1$ provas de reescrita (na variável *wBackList*) usando o também o método *checkList*, conforme pode ser visto na linha 14 do quadro C.2.

Quadro C.1: Protocolo do servidor (parte 1)

```
1   public AbstractMessage execute(NekoMessage aMessage){
2      AbstractMessage resp = null;
3      if(req instanceof MRQueryMessage){ //Se for uma mensagem de consulta
4         MRQueryMessage queryMsg = (MRQueryMessage)aMessage.getContent();
5         ReadWriteObject o = (ReadWriteObject)register.read();
6         if(queryMsg.isRead()){
7            MRQueryRespSignedMessage respSigned;
8            respSigned = new MRQueryRespSignedMessage(o,queryMsg.getNonce());
9            respSigned.setVerification(false); //cliente não verifica a mensagem
10           resp = respSigned;
11        }else
12           resp = new MRQueryRespMessage(o,queryMsg.getNonce());
13     }else if(req instanceof MRUpdateMessage){ //Se for uma mensagem de escrita
14        boolean written = false;
15        MRUpdateMessage uMsg = (MRUpdateMessage)aMessage.getContent();
16        ReadWriteObject updateObj = uMsg.getObject();
17        Long updateMsgObjTS = new Long(updateObj.getTimestamp());
18        ReadWriteObject o = (ReadWriteObject)this.echoedValues.get(updateMsgObjTS);
19        MRUpdateEchoSignedMessage respSigned;
20        /* Verifica se mensagem será ecoada... */
21        if(isTimestampOk(updateObj.getTimestamp(),aMessage.getSource())){
22           if(o != null){
23              if(o.compareValueTo(updateObj) == 0)
24                 respSigned = new MRUpdateEchoSignedMessage(updateObj,uMsg.getNonce());
25              else
26                 respSigned = new MRUpdateEchoSignedMessage(null,uMsg.getNonce());
27           }else{
28              this.echoedValues.put(updateMsgObjTS,updateObj);
29              respSigned = new MRUpdateEchoSignedMessage(updateObj,uMsg.getNonce());
30           }
31        }else
32           resp = new MRUpdateEchoSignedMessage(null,uMsg.getNonce());
33        respSigned.setVerification(false);
34        resp = respSigned;
35     }
36  }
```

Quadro C.2: Protocolo do servidor (parte 2)

```
1       else if(req instanceof MRUpdateListMessage){ //Se for uma mensagem de escrita pronta
2          MRUpdateListMessage updateListMsg = (MRUpdateListMessage)aMessage.getContent();
3          boolean written = false;
4          if(checkList(updateListMsg)){ //Verifica lista com um quórum de ecos assinados
5             ReadWriteObject o = (ReadWriteObject)register.read();
6             ReadWriteObject oToUpdate = updateListMsg.getObject();
7             written = true;
8             if(oToUpdate.compareTSTo(o) > 0) register.write(oToUpdate);
9          }
10         resp = new MRUpdateAckMessage(written,updateListMsg.getNonce());
11      }else if(req instanceof MRWriteBackListMessage){ //se for uma mensagem de reescrita
12         MRWriteBackListMessage wBackMsg = (MRWriteBackListMessage)aMessage.getContent();
13         boolean written = false;
14         if(checkList(wBackMsg)){ //Verifica lista com f+1 provas da reescrita
15            ReadWriteObject o = (ReadWriteObject)register.read();
16            ReadWriteObject oToUpdate = wBackMsg.getObject();
17            written = true;
18            if(oToUpdate.compareTSTo(o) > 0) register.write(oToUpdate);
19         }
20         resp = new MRUpdateAckMessage(written,wBackMsg.getNonce());
21      }
22      return resp;
23   }
```

Apêndice D

Exemplo de código-fonte no BQSNEKO – perfil bizantino

Este apêndice apresenta o código-fonte no BQSNEKO da implementação do perfil bizantino venenoso (*poisonous*), responsável por modificar valores de mensagens a serem enviadas (quadro D.1, ver página seguinte). Este perfil reimplementa o método *send* de um processo correto, escopo no qual realiza a sua ação faltosa através do método *poison*.

A execução do perfil é bem simples e se desenrola somente quando o processo que implementa o perfil faltoso envia uma mensagem do tipo AbstractObjectMessage (linha 27), classe genérica que implementa mensagens com um par $\langle v, t \rangle$. Além disto, para que efetivamente a mensagem seja modificada, o destinatário desta tem que ser um processo diferente de quem está realizando a ação faltosa (linha 34). Verificadas estas duas condições, o processo com perfil faltoso venenoso substitui o valor original da mensagem por um outro diferente. Na prática, o valor venenoso é o valor original somado ao identificador do processo destinatário da mensagem.

Quadro D.1: Implementação do perfil venenoso

```
1  package lse.neko.applications.bqs.layers.profile;
2
3  public class QPoisonousProfileLayer extends QProfileLayer{
4     public QPoisonousProfileLayer(NekoProcess process){
5        super(process,"pp");
6     }
7     private ReadWriteObject poison(ReadWriteObject in, int dest){
8        ReadWriteObject out;
9        if(in == null){
10          out = new ReadWriteObject();
11          out.setValue(dest);
12       }
13       else{
14          int value, newValue;
15          out = in;
16          value = ((out.getValue()==null)?dest:out.getValue().intValue());
17          newValue = value + dest;
18          out.setValue(newValue);
19       }
20       return out;
21    }
22    public void send(NekoMessage message){
23       AbstractMessage content = (AbstractMessage)message.getContent();
24       int from = message.getSource();
25       int[] to = message.getDestinations();
26       int type = message.getType();
27       if(content instanceof AbstractObjectMessage){
28          AbstractObjectMessage contentObj = (AbstractObjectMessage)content;
29          for(int i = 0; i < to.length; i++){
30             NekoMessage newMessage;
31             int[] newTo = {to[i]};
32             AbstractObjectMessage poisonContObj;
33             poisonContObj = (AbstractObjectMessage)contentObj.clone();
34             if(from != to[i])
35                poisonContObj.setObject(poison(poisonContObj.getObject(),to[i]));
36
37             NekoMessage poisonMessage;
38             poisonMessage = new NekoMessage(from,newTo,poisonContObj,type);
39             sender.send(poisonMessage);
40          }
41       }
42       else
43          sender.send(message);
44    }
45 }
```

Referências Bibliográficas

[1] Abd-El-Malek, M., Ganger, G., Goodson, G., Reiter, M., and Wylie, J. (2005). Fault-scalable Byzantine fault-tolerant services. Proc. *Proceedings of the 20th ACM Symposium on Operating Systems Principles - SOSP'05*, pp. 59–74.

[2] Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, Vol. 1, No. 1, pp. 11–33.

[3] Bakr, O. and Keidar, I. (2002). Evaluating the running time of a communication round over the internet. Proc. *PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pp. 243–252, New York, NY, USA. ACM Press.

[4] Barcellos, M., Woszezenki, C., and Munaretti, R. (2005). Framework de injeção de falhas simulada para avaliação de sistemas distribuídos. Proc. *Anais do 23o. Simpósio Brasileiro de Redes de Computadores - SBRC 2005*, Fortaleza, CE, Brasil.

[5] Bazzi, R. A. (2000). Synchronous byzantine quorum systems. *Distributed Computing*, Vol. 13, No. 1, pp. 45–52.

[6] Bazzi, R. A. and Ding, Y. (2004). Non-skipping timestamps for byzantine data storage systems. Proc. *Distributed Computing, 18th International Conference, DISC 2004, Amsterdam, The Netherlands, October 4-7, 2004, Proceedings*, Vol. 3274 of *Lecture Notes in Computer Science*, pp. 405–419. Springer.

[7] Ben-Or, M. (1983). Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). Proc. *Proceedings of the 2rd Annual ACM Symposium on Principles of Distributed Computing*, pp. 27–30.

[8] Bracha, G. and Toueg, S. (1985). Asynchronous consensus and broadcast protocols. *Journal of ACM*, Vol. 32, No. 4, pp. 824–840.

[9] Cachin, C. and Poritz, J. A. (2002). Secure intrusion-tolerant replication on the Internet. Proc. *Proceedings of the International Conference on Dependable Systems and Networks (DSN-2002)*, Washington, DC, USA. IEEE Computer Society Press.

[10] Cachin, C. and Tessaro, S. (2006). Optimal resilience for erasure-coded Byzantine distributed storage. Proc. *Proceedings of the International Conference on Dependable Systems and Networks - DSN 2006*.

[11] Castro, M. and Liskov, B. (2002). Practical Byzantine fault-tolerance and proactive recovery. *ACM Transactions Computer Systems*, Vol. 20, No. 4, pp. 398–461.

[12] Chandra, T. D. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, Vol. 43, No. 2, .

[13] Correia, M. (2005). Serviços distribuídos tolerantes a intrusões: resultados recentes e problemas abertos. Proc. *V Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais - Livro Texto dos Minicursos*, pp. 113–162. Sociedade Brasileira de Computação.

[14] Cristian, F., Aghali, H., Strong, R., and Dolev, D. (1985). Atomic broadcast: From simple message diffusion to Byzantine agreement. Proc. *Proceedings of the 15th International Symposium on Fault-Tolerant Computing (FTCS-15)*, pp. 200–206, Ann Arbor, MI, USA.

[15] Ekwall, R. and Schiper, A. (2005). Replication: Understanding the advantage of atomic broadcast over quorum systems. *Journal of Universal Computer Science*, Vol. 11, No. 5, pp. 703–711.

[16] Ekwall, R., Schiper, A., and Urbán, P. (2004). Token-based atomic broadcast using unreliable failure detectors. Proc. *Proc. 23nd IEEE Int'l Symp. on Reliable Distributed Systems (SRDS)*, pp. 52–65, Florianópolis, Brazil.

[17] Fischer, M. J., Lynch, N. A., and Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, Vol. 32, No. 2, pp. 374–382.

[18] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[19] Gifford, D. K. (1979). Weighted voting for replicated data. Proc. *Proceedings of the 7th ACM Symposium on Operating Systems Principles*, pp. 150–162.

[20] Goodson, G. R., Wylie, J. J., Ganger, G. R., and Reiter, M. K. (2004). Efficient byzantine-tolerant erasure-coded storage. Proc. *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks*, pp. 135, Washington, DC, USA. IEEE Computer Society.

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

[22] Herlihy, M. (1991). Wait-free synchronization. *ACM Transactions on Programing Languages and Systems*, Vol. 13, No. 1, pp. 124–149.

[23] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, Vol. 21, No. 7, pp. 558–565.

[24] Lamport, L. (1986). On interprocess communication (part ii: algorithms). *Distributed Computing*, Vol. 1, No. 1, pp. 203–213.

[25] Lamport, L. (1998). The part-time parliament. *ACM Transactions Computer Systems*, Vol. 16, No. 2, pp. 133–169.

[26] Lamport, L., Shostak, R., and Pease, M. (1982). The Byzantine generals problem. *ACM Transactions on Programing Languages and Systems*, Vol. 4, No. 3, pp. 382–401.

[27] Liskov, B. and Rodrigues, R. S. M. (2006). Tolerating byzantine faulty clients in a quorum system. Proc. *The 26th IEEE International Conference on Distributed Computing Systems - ICDCS 2006*.

[28] Malkhi, D. and Reiter, M. (1997). Unreliable intrusion detection in distributed computations. Proc. *Proceedings of the 10th Computer Security Foundations Workshop (CSFW97)*, pp. 116–124, Rockport, MA, USA.

[29] Malkhi, D. and Reiter, M. (1998a). Secure and scalable replication in Phalanx (extended abstract). Proc. *Proceedings of 17th Symposium on Reliable Distributed Systems*, pp. 51–60.

[30] Malkhi, D., Reiter, M., and Lynch, N. (1998). A correctness condition for memory shared by byzantine processes.

[31] Malkhi, D., Reiter, M., Tulone, D., and Ziskind, E. (2001). Persistent objects in the fleet system.

[32] Malkhi, D. and Reiter, M. K. (1998b). Byzantine quorum systems. *Distributed Computing*, Vol. 11, No. 4, pp. 203–213.

[33] Martin, J.-P. and Alvisi, L. (2005). Fast Byzantine consensus. Proc. *Dependable Systems and Networks, DSN 05*.

[34] Martin, J.-P., Alvisi, L., and Dahlin, M. (2002a). Minimal Byzantine storage. Proc. *Distributed Computing, 16th international Conference, DISC 2002*, Vol. 2508 of *LNCS*, pp. 311–325.

[35] Martin, J.-P., Alvisi, L., and Dahlin, M. (2002b). Small byzantine quorum systems. Proc. *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pp. 374–388, Washington, DC, USA. IEEE Computer Society.

[36] Moniz, H., Neves, N. F., Correia, M., and Veríssimo, P. (2006). Randomized intrusion-tolerant asynchronous services. Proc. *Proceedings of the International Conference on Dependable Systems and Networks - DSN 2006*.

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

[38] Rabin, M. O. (1983). Randomized Byzantine generals. Proc. *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science*, pp. 403–409.

[39] Rabin, M. O. (1989). Efficient dispersal of information for security, load balancing, and fault tolerance. *J. ACM*, Vol. 36, No. 2, pp. 335–348.

[40] Reiter, M. K. (1994). Secure agreement protocols: Reliable and atomic group multicast in rampart. Proc. *Proceedings of the 2nd ACM Conference on Computer and Communication Security*, pp. 68–80.

[41] Schneider, F. B. (1990). Implementing fault-tolerant service using the state machine aproach: A tutorial. *ACM Computing Surveys*, Vol. 22, No. 4, pp. 299–319.

[42] Thomas, R. H. (1979). A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, Vol. 4, No. 2, pp. 180–209.

[43] Tindell, K., Burns, A., and Wellings, A. J. (1995). Analysis of hard real-time communications. *Real-Time Syst.*, Vol. 9, No. 2, pp. 147–171.

[44] Urbán, P., Défago, X., and Schiper, A. (2001). Chasing the FLP impossibility result in a LAN or how robust can a fault tolerant server be? Proc. *Proc. 20th IEEE Symp. on Reliable Distributed Systems (SRDS)*, pp. 190–193, New Orleans, LA, USA.

[45] Urbán, P., Hayashibara, N., Schiper, A., and Katayama, T. (2004). Performance comparison of a rotating coordinator and a leader based consensus algorithm. Proc. *Proc. 23nd IEEE Int'l Symp. on Reliable Distributed Systems (SRDS)*, pp. 4–17, Florianópolis, Brazil.

[46] Urbán, P., Défago, X., and Schiper, A. (2000). Contention-aware metrics for distributed algorithms: Comparison of atomic broadcast algorithms. Proc. *Proceedings of the 9th IEEE Int'l Conference on Computer Communications and Networks (IC3N 2000)*.

[47] Urbán, P., Défago, X., and Schiper, A. (2001). Neko: A single environment to simulate and prototype distributed algorithms. Proc. *Proceedings of the 15th Int'l Conf. on Information Networking (ICOIN-15)*, Beppu City, Japan.

[48] Zhou, L., fred B. Schneider, and Van Rennesse, R. (2002). COCA: A secure distributed online certification authority. *ACM Transactions Computer Systems*, Vol. 20, No. 4, pp. 329–368.

[49] Zielinski, P. (2004). Paxos at war. Relatório Técnico UCAM-CL-TR-593, University of Cambridge Computer Laboratory, Cambridge, UK.