Um Arcabouço de Simulação e Avaliação de Sistemas de Quóruns Bizantinos

Wagner Saback Dantas[1] **, Alysson Neves Bessani**[2] **, Joni da Silva Fraga**[1]

[1]Departamento de Automação e Sistemas
Universidade Federal de Santa Catarina
Florianópolis - SC

[2]LaSIGE - Laboratório de Sistemas de Grande Escala
Faculdade de Ciências da Universidade de Lisboa
Lisboa - Portugal

{wagners,fraga}@das.ufsc.br,neves@lasige.di.fc.ul.pt

Abstract. *Byzantine Quorum Systems (BQS) have been described as a good choice to build Byzantine-tolerant reliable distributed storage, with many implementation approaches already proposed. Choosing the most suited one requires a careful evaluation aided by tools oriented to model and prototype Byzantine execution environments. To date, however, such facility does not exist. The paper presents BQSNeko, a framework useful for evaluating BQS algorithms. To show how this framework can be used, we discuss the results of an experiment built on* BQSNEKO *that compares two well-known Byzantine quorum protocols.*

Resumo. *Sistemas de Quóruns Bizantinos (BQS) têm sido apresentados como uma boa estratégia de construção de armazenamento distribuído tolerante a faltas bizantinas, havendo muitas abordagens para sua implementação. Escolher a abordagem mais adequada requer uma avaliação minuciosa com a ajuda de ferramentas orientadas à modelagem e prototipação de ambientes bizantinos de execução. Até então, porém, não se conhecia uma ferramenta que facilitasse tal análise. O artigo apresenta o arcabouço de avaliação de algoritmos de BQS* BQSNEKO. *Para mostrar a sua utilidade, serão discutidos os resultados de um experimento construído no* BQSNEKO *com dois algoritmos de BQS conhecidos.*

1 Introdução

Sistemas de Quóruns Bizantinos (*Byzantine Quorum Systems* ou BQS) [Malkhi and Reiter 1998b] são um meio de se obter garantias de consistência e disponibilidade em armazenamento de dados replicados mesmo com a ocorrência de faltas bizantinas [Lamport et al. 1982]. Em BQS, dados são replicados em diferentes conjuntos de servidores (**quóruns**) que compartilham um número suficiente de servidores em comum. Clientes lêem e escrevem no sistema somente por meio desses quóruns, respectivamente em quóruns de leitura e escrita. Assim, diferentes operações do cliente podem ser executadas em diferentes quóruns sem perda de consistência, colaborando com a escalabilidade e o bom desempenho do sistema. Outro ponto favorável à implementação eficiente de armazenamento com BQS vem do não uso de acordo para manter consistência de dados, que é garantida pela própria interseção dos quóruns.

Na literatura, muitas abordagens de implementação de BQS já foram propostas. Estas soluções refletem diferentes perspectivas de projeto na construção de um sistema de armazenamento usando BQS de acordo com características chaves como o tamanho dos quóruns de leitura e escrita, o modelo de falhas dos clientes e a semântica de consistência de dados suportada. A escolha de qual das abordagens seguir passa por uma avaliação minuciosa de qual algoritmo para 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. Entretanto, não se conhecia uma ferramenta que contemplasse tais tarefas.

Este trabalho apresenta o BQSNEKO, uma aplicação extensível desenvolvida sobre o arcabouço NEKO [Urbán et al. 2001] útil na análise de protocolos de BQS. Aproveitando-se das funcionalidades providas pelo NEKO, o BQSNEKO permite a simulação, a execução e, posteriormente, a avaliação de protocolos de BQS, considerando seus aspectos, como a ausência de tempo nos algoritmos e a simplicidade no lado servidor. Ambas simulação e execução dos protocolos podem considerar cenários de ataques a partir da injeção de faltas bizantinas no sistema. O perfil de falta bizantina pode já ser oferecido pelo BQSNEKO ou possivelmente implementado usando facilidades oferecidas pelo arcabouço. A implementação de algoritmos de BQS torna-se muito mais simples usando o BQSNEKO uma vez que várias tarefas necessárias para construir estes algoritmos já são suportadas pelo próprio BQSNEKO.

A fim de demonstrar o uso do BQSNEKO na avaliação de algoritmos de BQS, ilustraremos o desenvolvimento de um experimento que simula as operações de escrita em BQS de dois algoritmos conhecidos em dois ambientes de rede distintos e avalia de que forma os dois algoritmos comportam-se face a um serviço de armazenamento com alguns servidores bizantinos: tais servidores falham alterando os valores de uma mensagem tentando comprometer a consistência do sistema.

O artigo está organizado da seguinte forma: a seção 2 apresenta uma visão geral sobre Sistemas de Quóruns Bizantinos. A seção 3 descreve a arquitetura geral do NEKO (seção 3.1) e detalhada do BQSNEKO (seção 3.2), explicando as suas funcionalidades principais: como desenvolver algoritmos de BQS e perfis de falta bizantina (seção 3.3) e como executar esses algoritmos (seção 3.4). A seção 4 aborda o experimento de exemplo. A seção 4.1 especifica os componentes teóricos envolvidos (BQS, algoritmos e mecanismos usados), a seção 4.2 descreve as configurações do ambiente, e a seção 4.3 relata os resultados coletados e a análise desses resultados propriamente. As seções 5 e 6 apresentam alguns trabalhos relacionados e as conclusões finais do trabalho, respectivamente. O apêndice A exibe o exemplo-base do arquivo de configuração do BQSNEKO utilizado na realização dos experimentos da seção 4.

2 Sistema de Quóruns Bizantinos

Sistemas de quóruns Bizantinos (BQS) [Malkhi and Reiter 1998b] são uma técnica de replicação que garante as propriedades de consistência (os dados são acessados corretamente) e disponibilidade (os dados sempre podem ser acessados) em sistemas de armazenamento distribuídos mesmo na existência de processos que falham de maneira arbitrária. Ao contrário dos sistemas de armazenamento baseados no paradigma de Replicação Máquinas de Estado (RME) [Lamport 1978, Schneider 1990], a implementação de BQS

não requer acordo entre as réplicas do serviço, uma vez que estes conseguem emular, no máximo, registradores atômicos [Lamport 1986]. Tais objetos compartilhados suportam apenas operações de leitura e escrita e podem ser construídos sem o uso de acordo [Herlihy 1991]. Por este motivo, quando se tratam de sistemas assíncronos, BQS não são suscetíveis à impossibilidade FLP [Fischer et al. 1985].

Estes sistemas apresentam também uma boa escalabilidade tanto pelo fato de os clientes acessarem somente uma parte dos servidores do sistema (um quórum) na execução de uma operação quanto pelos servidores executarem, em geral, tarefas simples, deixando a parte mais complexa dos protocolos para o lado cliente.

No modelo de sistema de BQS usualmente definido [Malkhi and Reiter 1998b], os processos se comunicam em um modelo assíncrono por passagem de mensagens em canais ponto a ponto confiáveis e autenticados, sendo que existe um conjunto U com n servidores e um conjunto de clientes Π disjunto de U e possivelmente ilimitado. Os servidores se organizam em subconjuntos de U chamados **quóruns**, onde, para quaisquer dois quóruns, existe um número suficiente de servidores corretos na sua interseção (garantia de consistência). Além disso, num sistema de quóruns, um quórum é formado apenas por servidores corretos pelo menos (garantia de disponibilidade). Cada processo servidor é um repositório de dados que armazena uma cópia local de uma variável (registrador). Os clientes realizam operações de leitura e escrita nestas variáveis acessando quóruns de leitura e escrita de mesmo tamanho (**quóruns simétricos**) ou tamanhos diferentes (**quóruns assimétricos**).

Em termos de falhas de processos, assume-se que uma quantidade de servidores pode apresentar faltas bizantinas [Lamport et al. 1982], isto é, podem desviar-se arbitrariamente de seu algoritmo e executar qualquer tipo de ação (maliciosa ou não) no sistema. Os algoritmos supõem que até f servidores podem ser corrompidos (faltosos). A relação entre f e n depende do algoritmo e modelo de sistema empregados nos BQS. Diferentes algoritmos assumem diferentes modelos de falhas nos clientes. Dentre as premissas mais comuns estão: clientes corretos (nenhuma falta é tolerada em clientes), clientes podem falhar por parada e clientes estão sujeitos a faltas bizantinas.

Formalmente, uma variável em BQS é implementada através da abstração de um **registrador** x com suporte a operações de leitura e escrita. Cada servidor armazena uma cópia de x, representado pelo par $\langle v, t \rangle$, onde v é o valor atual armazenado e t é a estampilha de tempo (*timestamp*) associada a v. Existem diversas semânticas para registradores conforme a hierarquia de Lamport [Lamport 1986]. Cada semântica define de maneira particular o comportamento de um registrador segundo a ocorrência de leitura e escrita concorrentes e a quantidade de leitores e escritores suportados pelo algoritmo.

Muitos protocolos têm sido propostos no contexto de sistemas de quóruns Bizantinos (e.g., [Malkhi and Reiter 1998b, Malkhi and Reiter 1998a, Martin et al. 2002a, Liskov and Rodrigues 2006]). Suas construções diferenciam-se, por exemplo, pela organização dos quóruns (tamanho e simetria dos quóruns de leitura e escrita), pela semântica de consistência dos seus registradores replicados ou pelo modelo de falhas de seus clientes. Essa variedade é justificável visto que algoritmos com diferentes propriedades conseguem trabalhar eficientemente sob diferentes pontos de vista em diferentes ambientes de execução com variadas exigências.

3 Arcabouço BQSNEKO

Esta seção descreve o arcabouço BQSNEKO e suas funcionalidades. Antes, porém, será dada uma visão geral do NEKO, base para o BQSNEKO. Depois, serão mostradas como foram feitas as extensões no NEKO para implementar algoritmos de BQS e os perfis bizantinos (para simulação de cenários de ataques) que dão origem ao BQSNEKO.

3.1 NEKO

NEKO [Urbán et al. 2001] é um arcabouço escrito em Java para prototipação e avaliação de algoritmos distribuídos em redes simuladas ou reais. Na arquitetura do NEKO, uma aplicação é constituída de 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 (reais ou simuladas).

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). A camada mais inferior da aplicação comunica-se com o processo NEKO que, por sua vez, envia e coleta mensagens da rede. Camadas podem ser passivas ou ativas: numa camada passiva, as mensagens são conduzidas indiretamente pela sua camada inferior usando o método *deliver*; numa camada ativa, mensagens são diretamente conduzidas usando o método *receive*, que devolve uma mensagem previamente recebida e armazenada numa fila de recepção.

3.2 Arquitetura do BQSNEKO

Considerando a implementação de algoritmos de BQS, o NEKO apresenta, pelo menos, duas limitações: *(i)* ausência de um mecanismo para injeção de faltas bizantinas; *(ii)* ausência de um *framework* para implementação de algoritmos de BQS que tire proveito das similaridades desse tipo de algoritmo. Deste modo, a fim de prover um melhor suporte à prototipação e avaliação dessa classe de algoritmos e com vistas ao seu ambiente de execução, o BQSNEKO surge como uma extensão ao NEKO.

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

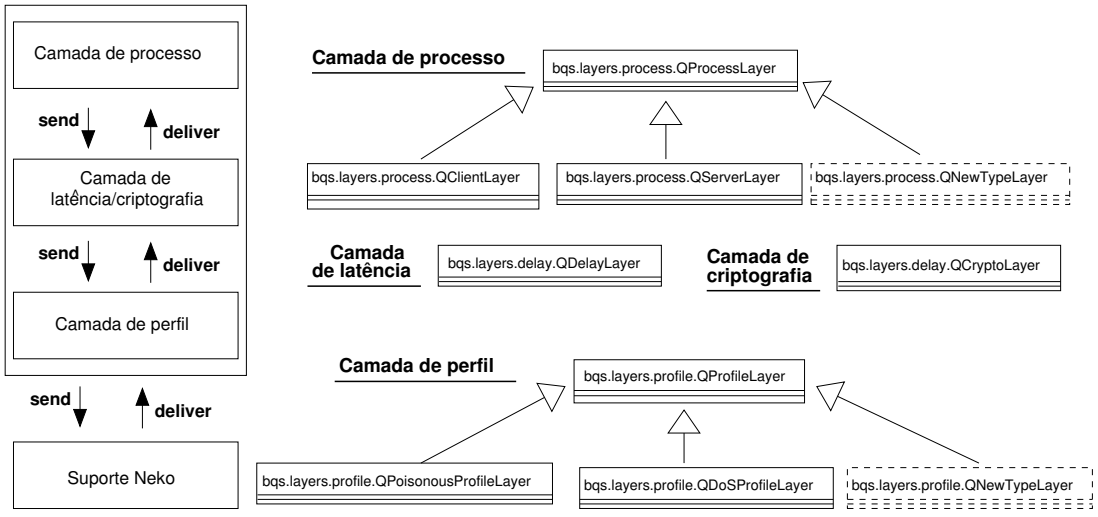
1. **Informações de configuração**: descrevem as características básicas do Sistema de Quóruns Bizantinos usado e os seus parâmetros de configuração (e.g., número de processos no sistema e tamanho dos quóruns de leitura e escrita). No BQSNEKO, essas informações estão contidas num objeto de dados apropriado e são usadas na 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. Tal processo é composto por 3 camadas (2 passivas e 1 ativa) conforme apresentado na Figura 1(a):

[1] Até a versão 0.9 (usada pelo BQSNEKO), é usada organização em camadas. A versão atual (1.0) utiliza outro modelo de organização.

(a) **Camada de processo**: camada ativa de um processo genérico de BQS. Implementa camadas genéricas do cliente ou servidor em um BQS. Os algoritmos cliente e servidor de um sistema de quóruns também são implementados nesta camada;

(b) **Camada de latência/criptografia**: usadas para aplicar o custo adicional de processamento no envio e recepção de mensagens segundo o modelo de sistema de BQS e de acordo com o protocolo desenvolvido. Esse custo contabiliza, por exemplo, o emprego de operações criptográficas. Na simulação, este efeito pode ser gerado pela camada de latência de acordo com valores de atraso possivelmente definidos para alguns tipos de mensagem do protocolo. Isto é útil para refinar a atribuição de custos de processamento local aos modelos de rede que não consideram adequadamente o tempo de CPU nos processos em simulação. Estes valores são especificados nos parâmetros de configuração do experimento. Na execução em rede, o atraso é realizado pela camada de criptografia, que aplica operações criptográficas implementadas pela JCE (*Java Cryptography Extensions*) em algumas mensagens segundo a especificação do protocolo de BQS.

(c) **Camada de perfil**: define o perfil de falha do processo. O processo pode ser correto (seguindo os algoritmos implementados) ou faltoso (desviando-se arbitrariamente do comportamento esperado).



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

Figura 1. Modelos de camadas de um processo BQSNeko

3.3 Prototipando com o BQSNeko

Implementando um novo algoritmo para um sistema de quóruns Bizantinos. Conceitualmente, antes de construir um novo algoritmo, é preciso especificar a estrutura de quóruns explorada pelo algoritmo, bem como o acervo de mensagens a serem trocadas entre processos. No BQSNeko, a estrutura de quóruns é especificada por um objeto de dados que define as informações do sistema de quóruns utilizado. Já as mensagens usadas pelos protocolos de BQS são implementadas por classes genéricas de mensagens, o mesmo valendo para as partes cliente e servidor do protocolo.

Para definir um objeto de dados BQSNEKO e as novas mensagens dos protocolos do sistema implementado, o arcabouço provê duas classes genéricas: uma de objetos de dados e outra de mensagens trocadas durante a execução de algoritmos de BQS (classes QAbstractInfo e AbstractMessage, respectivamente). A classe QAbstractInfo recebe como argumento o limite máximo de faltas no sistema e guarda informações essenciais, como o número mínimo de servidores no sistema e os tamanhos mínimos dos quóruns de leitura e escrita. A classe AbstractMessage mantém atributos importantes de uma mensagem, como o tipo de mensagem. A partir destas duas classes, respectivamente, podem ser criados novos objetos de dados com informações de sistemas de quóruns específicas e novos tipos de mensagens.

Os protocolos para o cliente e o servidor são definidos na camada de processo do BQSNEKO. Esta camada oferece classes genéricas (figura 1(b)) do cliente (QClientLayer) e servidor (QServerLayer) que definem métodos abstratos *read* e *write* para operações do cliente e o método *execute* para execuções do servidor, respectivamente. Estas classes genéricas devem ser estendidas para criar as camadas de processo onde residem os protocolos cliente e servidor que executam sobre a estrutura de sistema de quóruns especificada no objeto de dados BQSNEKO.

Simulando operações criptográficas (execuções em rede real somente). Implementar uma nova mensagem do protocolo sobre a qual se usam operações de assinatura (no envio) e verificação criptográfica (na recepção) requer que sua classe correspondente estenda AbstractChallengeMessage. Por padrão, a mensagem implementada desta maneira terá os seus custos de assinatura e verificação ativados pela camada de criptografia quando do algoritmo executado em rede, respectivamente durante cada envio e recepção da mesma. Para desativar o custo de assinatura (quando se sabe que o emissor não assina a mensagem), utiliza-se 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 falta bizantino, é preciso primeiro estender a classe genérica QProfileLayer (figura 1(b)) da camada de perfil da arquitetura BQSNEKO, definindo um novo método *send* com o comportamento do processo faltoso. Um exemplo de comportamento faltoso poderia ser o seguinte: se um processo está sofrendo um ataque de negação de serviço (DoS), então ele demora *k* vezes mais tempo para responder a uma requisição, sendo *k* um parâmetro configurável. Por padrão, a classe genérica da camada de perfil já define o método *send* como na execução de um processo correto. A versão atual do BQSNEKO implementa dois perfis bizantinos: um perfil venenoso (classe QPoisonousProfileLayer, que ocasiona falha por valor), que altera valores de mensagens, e um perfil DoS (classe QDoSProfileLayer, que ocasiona falha temporal) de acordo com o cenário de ataque descrito anteriormente. Ambas as classes estão ilustradas na Figura 1(b).

Note que, na prática, as falhas não são injetadas nos processos, mas no canal de comunicação durante o envio da mensagem (em termos de implementação, como dito, isto significa modificar a primitiva *send* do processo). Este modelo, onde canais podem corromper (ou omitir) as mensagens arbitrariamente, equivale ao modelo de processos bizantinos, visto que todo comportamento malicioso pode ser representado, sendo que suas implementação e configuração dentro do arcabouço são bem simples.

3.4 Executando algoritmos de BQS

Definindo o ambiente de execução. No BQSNEKO, o ambiente de execução de um algoritmo de BQS é definido a partir de um arquivo de configuração estendido do NEKO. Este arquivo estendido está dividido em duas partes: a primeira define as configurações genéricas de uma execução, próprias do NEKO, como, por exemplo, se a execução ocorrerá sobre uma rede simulada ou real, quantos processos existirão na execução e qual o objeto responsável pela iniciação do ambiente de execução (neste caso, o parâmetro tem valor fixo, a classe BQSInitializer); a segunda parte das configurações diz respeito aos parâmetros específicos dos algoritmos de BQS, como, por exemplo, o limite máximo de faltas bizantinas, o tamanho padrão dos quóruns de leitura e escrita, etc. A versão atual do BQSNEKO já implementa os principais algoritmos de BQS conforme mostrados pela tabela 1.

Referência	**Quóruns**	**Semântica** [Lamport 1986]	**Clientes**
[Malkhi and Reiter 1998b]	simétricos	*MWMR*[1] *safe/regular* *SWMR*[2] *safe*	corretos bizantinos
[Malkhi and Reiter 1998a]	simétricos	*MWMR atomic* *MWMR safe*	corretos bizantinos
[Martin et al. 2002a]	assimétricos	*MWMR atomic*	corretos bizantinos
[Liskov and Rodrigues 2006]	simétricos	*MWMR atomic*	bizantinos

Tabela 1. Algoritmos de BQS implementandos no BQSNEKO

Definindo os protocolos de BQS e configurações associadas. Os protocolos cliente e servidor de um sistema de quóruns são definidos pelos parâmetros qclientlayer e qserverlayer, respectivamente. 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 suportam faltas em clientes. O tempo atribuído na camada de latência a um processo (seção 3.2) é descrito por um parâmetro no formato latency.**message-type**.**event**, que define um custo adicional que um processo terá com uma mensagem de um tipo definido **message-type** durante seu envio (**event** = *send*) e recepção (**event** = *receive*).

Definindo perfis bizantinos. 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 maneira simples e extensível usando o parâmetro de formato faulty.**process-type**.**profile-classname**.percent que define a quantidade percentual aproximada (arredondando o valor para cima) 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 executam o perfil de faltas implementado pela classe Java QDoSProfileLayer.

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

Definindo a execução do cliente. A execução do cliente é definida pelas classes TestReadClient (operações de leitura) e TestWriteClient (operações de escrita). No arquivo de configuração da execução, o parâmetro layer.application.type.**client-ID** indica o tipo de operação a ser efetuada no sistema de quóruns (*read* ou *write*) pelo processo **client-ID**. O número de vezes em que se realiza uma operação do tipo **operation-type** (*read* ou *write*) é designado pelo parâmetro layer.application.executions.**operation-type**.

4 Exemplo de Experimento

Esta seção descreve um exemplo de experimento de avaliação de algoritmos de BQS que pode ser feita com o BQSNEKO. O objetivo do experimento é avaliar dois mecanismos de consistência empregados respectivamente por dois algoritmos de BQS conhecidos da literatura.

O conteúdo desta seção inicialmente descreve o experimento em questão, as propriedades dos sistemas de quóruns explorados e dos algoritmos envolvidos, bem como as suas respectivas técnicas. Depois, é apresentada a especificação do ambiente do experimento, colocando mais detalhes sobre os modelos de rede assumidos e as cargas consideradas nas execuções dos algoritmos. Por último, a seção expõe as análises dos resultados coletados e, segundo as premissas observadas no experimento, reflete sobre algumas possibilidades de uso dos protocolos testados.

4.1 Descrição

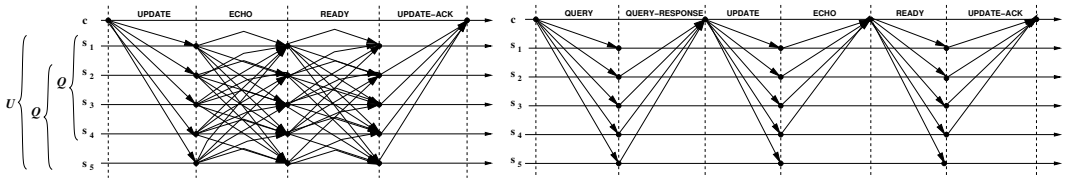
O experimento consiste na comparação das latências associadas ao uso de dois métodos de consistência empregados pelos algoritmos em avaliação a partir de medições dos tempos de execução da operação de escrita de 1 cliente correto. Este caso observa dois algoritmos extraídos da literatura de BQS, que têm propriedades similares e que são desempenhados sobre dois BQS com propriedades idênticas. Os resultados mostram valores de tempo de execução de operações de escritas consolidadas no sistema (isto é, em um quórum de escrita) em cenários variados com apenas servidores corretos e com alguns servidores bizantinos.

Os BQS explorados pelos algoritmos em questão possuem construções idênticas com quóruns simétricos $Q = 3f + 1$ servidores e com um sistema de réplicas U com $|U| \geq 4f + 1$ servidores. Cada dois quóruns neste caso mantêm em sua interseção $2f + 1$ servidores, o que garante que uma maioria de servidores possua um valor correto. Por outro lado, os protocolos de escrita neste sistema de quóruns empregam técnicas diferentes para garantir a consistência do valor escrito em face a possibilidade de clientes faltosos.

O primeiro algoritmo [Malkhi and Reiter 1998b], chamado simplesmente de "*reliable*", emprega difusão confiável de mensagens entre os servidores para assegurar que um valor escrito em um servidor seja escrito ao menos em um quórum de servidores corretos. O algoritmo *reliable* executa em 4 passos e gera um conjunto de mensagens na ordem de $O(n^2)$. Garante semântica de consistência segura e semântica de acesso "simples escritor, múltiplos leitores". Sua execução está ilustrada pela figura 2(a).

O segundo algoritmo [Malkhi and Reiter 1998a], chamado apenas de "*echo*", usa um protocolo de difusão com eco para evitar que um cliente malicioso escreva diferentes valores em diferentes servidores. Para isso, emprega 6 passos na sua execução com complexidade de mensagens na ordem de $O(n)$. Este algoritmo garante também uma

semântica de consistência segura, porém uma semântica de acesso “múltiplo escritor, múltiplos leitores”. Todos estas propriedades podem ser vistas na tabela 1 da seção anterior. Sua execução está ilustrada pela figura 2(b).



(a) Algoritmo *reliable* (b) Algoritmo *echo*

Figura 2. Algoritmos de escrita em quórum bizantino experimentados

4.2 Configurações do Ambiente

As simulações ocorrem sobre um **Modelo de Rede Simulada com Noção de Contenção (*Contention-Aware Simulated Network Model*)** [Urbán et al. 2000], já implementado pelo NEKO. Esse modelo de rede considera um único parâmetro de entrada $\lambda \geq 0$ que define o desempenho relativo entre os recursos CPU e rede. Para modelagem de redes locais, considera-se normalmente um $\lambda = 10$ (contenção maior na CPU) e, para redes de larga escala, um $\lambda = 0,1$ (contenção maior na rede), conforme verificado em trabalhos similares como [Urbán et al. 2004].

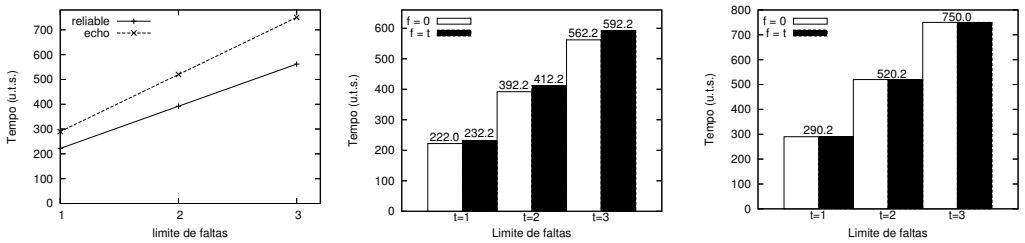
Todos os servidores faltosos podem agir como processos que tentam comprometer a consistência do sistema alterando os valores de suas mensagens enviadas (servidores venenosos, parâmetro faulty.server.QPoisonousProfileLayer.percent = 100). O limite de faltas t no sistema varia entre 1 e 3. Para cada limite t, assume-se um número f de servidores bizantinos no sistema, onde $0 \leq f \leq t$. Neste experimento, consideramos dois cenários de falhas apenas. No primeiro, todos os servidores são corretos ($f = 0$). No segundo, alguns servidores são bizantinos, ocasião na qual o número de servidores bizantinos é sempre o máximo estabelecido pelo limite de faltas, isto é, $f = t$. Considera-se em todas as execuções o limite mínimo de servidores, ou seja, $n = 4f + 1$. Os clientes executam operações no sistema apenas uma vez (layer.application.executions.write = 1) por se tratar de um modelo de simulação, onde não ocorreriam variações nos tempos de execução caso fossem feitas múltiplas operações seguidas.

4.3 Resultados

Os gráficos das figuras 3 e 4 apresentam os resultados da simulação considerando redes locais ($\lambda = 10$, figura 3) e de larga escala ($\lambda = 0,1$, figura 4). Importante notar que os resultados não são propriamente os valores dos tempos de execução obtidos (tempo simulado), mas o comportamento dos algoritmos refletido das variações desses tempos coletados conforme informam os gráficos tanto para o caso com servidores corretos (figuras 3(a) e 4(a)) como no impacto de adição de faltas em alguns servidores no sistema (figuras 3(b), 3(c), 4(b) e 4(c)), respectivamente. A partir destas variações, é possível ter uma noção da escalabilidade e do desempenho dos protocolos quando do aumento da expectativa de limite de faltas e da adição de servidores efetivamente bizantinos.

No cenário de rede local, dentro do intervalo de limite de faltas simulado e com servidores corretos (figura 3(a)), percebe-se um melhor desempenho no uso da difusão entre servidores (algoritmo *reliable*) em relação à difusão com eco (algoritmo *echo*). O fato

se explica pela maior quantidade de passos de execução no algoritmo *echo* (6 passos contra 4 do algoritmo *reliable*) e pelo seu uso de assinatura criptográfica, sobretudo, durante a etapa de "eco", onde os servidores recebem o valor a ser escrito pelo cliente, assinam este valor e o devolvem ao cliente. Se for observado que o modelo de rede em questão considera maior contenção de CPU ($\lambda = 10$), tem-se que a maior carga de operações locais no algoritmo *echo* (leia-se, a criptografia envolvida na etapa de eco), acaba tendo maior impacto no resultado final do que a transmissão de mensagens entre servidores no algoritmo *reliable*.



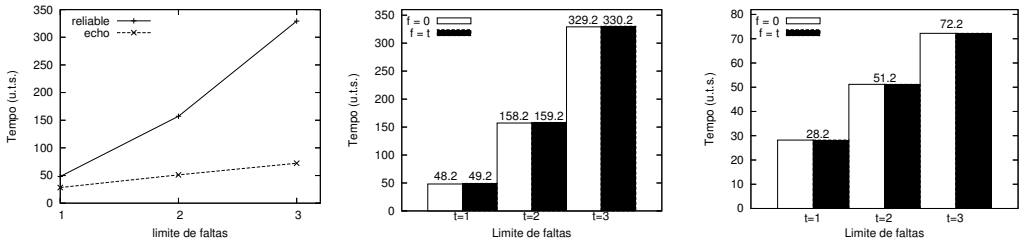
(a) Sem faltas (b) Algoritmo *reliable* (c) Algoritmo *echo*

Figura 3. Experimentos em LAN simulada

Esta perspectiva se mantém quando são injetadas faltas no experimento. Entretanto, no algoritmo *echo* não há variações no tempo total de execução do cliente, ao contrário do algoritmo *reliable*. Realmente, ao se observar a descrição de execução do algoritmo *reliable*, percebe-se que o cliente é prejudicado por um atraso na resposta do quórum de escrita, dentro do qual cada servidor correto, ao longo de suas etapas de execução, descarta os valores enviados pelos servidores bizantinos, culminando em uma maior demora para reunir um quórum de valores somente de servidores corretos capaz de retornar a operação para o cliente. Por outro lado, o cliente no algoritmo *echo* não realiza a verificação de assinaturas recebidas de um quórum de servidores (cliente pode ser bizantino), que fica a cargo dos servidores somente. No ponto de vista do cliente, o tempo de execução é o mesmo, pela própria imutabilidade no total de suas ações (clientes não precisam esperar por mais mensagens já que não fazem verificação) e pelo fato de o número de operações locais dos servidores (assinaturas e verificações, sobretudo) também ser o mesmo. O que poderia diferenciar as latências de execução do cliente seria a variação dos tempos de transmissão de mensagens, o que é ínfima em redes locais na prática e no modelo de simulação empregado.

Na rede de larga escala experimentada, observou-se um cenário inverso ao obtido no modelo de rede local. Com apenas servidores corretos (figura 4(a)), à medida que o limite de faltas aumenta, o desempenho do algoritmo *reliable* degrada-se mais do que o do algoritmo *echo*. Dessa forma, dentro das mesmas premissas do experimento, o comportamento dos algoritmos vai de encontro às suas propriedades teóricas: o algoritmo *reliable* possui uma complexidade de troca de mensagens de $O(n^2)$ (crescimento exponencial), e o algoritmo *echo*, de $O(n)$ (crescimento linear). Esta inversão ocorre porque no ambiente de larga escala ($lambda = 0,1$), a contenção é normalmente maior na comunicação entre processos e não no custo local de processamento. Como o algoritmo *reliable*, a despeito do número menor de passos de execução, utiliza em seu mecanismo de consistência a troca de mensagens entre todos os servidores, o que significa um conjunto

global maior de mensagens gerado, o impacto apresenta-se maior do que o percebido no algoritmo *echo*, cujo maior custo, as operações criptográficas conforme visto, é diluído na contagem total da latência de execução do cliente.



(a) Sem faltas (b) Algoritmo *reliable* (c) Algoritmo *echo*

Figura 4. Experimentos em WAN simulada

Quando da adição de uma fração de servidores como bizantinos igual ao limite de faltas do sistema, as variações incidem apenas sobre o algoritmo *reliable*. Isto equivale ao tempo de espera por mais uma mensagem de um processo correto, provocado pelo anterior descarte de uma mensagem vinda de um servidor bizantino. Note, porém, que esta variação é praticamente desprezível se comparada ao desempenho total do algoritmo. Esta mudança não ocorre no algoritmo *echo*, que não apresenta mudanças de desempenho final. Como explicado, o cliente que executa o algoritmo *echo* não realiza operações criptográficas, tampouco os servidores alteram o seu padrão de execução mesmo com a influência de servidores bizantinos. De fato, o desempenho do cliente poderia apresentar variação na prática, caso o experimento fosse feito em ambiente de larga escala real, com características intrínsecas de variações de tempos de transmissão, o que não se aplica ao modelo de simulação em questão.

Com base nos resultados obtidos, tendo o devido cuidado de guiá-los pelas configurações de experimento adotadas, isto é, supondo poucas réplicas no serviço por conta da dificuldade prática de se implementar sistemas com muitas réplicas e independência de falhas [Obelheiro et al. 2005], podemos ponderar algumas conclusões. A primeira delas é que o uso do algoritmo *reliable* não é aconselhável em redes de larga escala, onde a contenção é maior na rede. Em redes locais, o algoritmo *echo* acaba sendo prejudicado no contexto de baixo número de réplicas, ainda que, em conceito, tenha complexidade linear ao passo que o algoritmo *reliable* tenha complexidade quadrática. Neste caso, observa-se que ambos os algoritmos demonstram um progresso linear em seu tempo de execução. Em todos os cenários observados, a injeção de faltas não foi expressivamente custosa, o que demonstra uma boa resistência dos dois protocolos ao comparecimento de servidores bizantinos (ao menos, servidores bizantinos com perfis de “servidores venenosos”) dentro do limite de faltas esperado.

5 Trabalhos Relacionados

O ambiente NEKO [Urbán et al. 2001] 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 (*crashing*), por exemplo [Urbán et al. 2004].

No contexto de BQS, embora haja muitas propostas de algoritmos (por exemplo, [Malkhi and Reiter 1998b, Malkhi and Reiter 1998a, Martin et al. 2002a, Martin et al. 2002b, Liskov and Rodrigues 2006]), não se pôde notar trabalhos que analisem diferentes algoritmos de BQS (somente análises específicas). Poucos trabalhos analisam os seus algoritmos tendo em vista cenários bizantinos: [Martin et al. 2002a] propõe o algoritmo SBQ-L tolerante a faltas bizantinas e apresenta uma avaliação deste algoritmo, porém sua análise não considera a ocorrência de faltas; [Goodson et al. 2004] compara sua abordagem de consistência em sistemas replicados de armazenamento Bizantino usando BQS com a abordagem baseada em replicação por máquina de estados [Lamport 1978, Schneider 1990] e não considera 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 [Urbán et al. 2001]), mas com distintos focos. Um caso particular é o arcabouço Simmcast-FT [Barcellos et al. 2005]: 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.

6 Considerações Finais

O BQSNEKO pode ser uma ferramenta útil para avaliação de protocolos de BQS e posterior escolha de qual algoritmo adequa-se melhor a determinado ambiente de execução. Este trabalho, além de apresentar este arcabouço, ilustra como ele pode ser usado na análise de algoritmos de BQS através de um exemplo de experimento comparando dois algoritmos conhecidos da literatura com propriedades similares, desempenhados sobre construções de quóruns similares, mas com técnicas diferentes, evidenciando uma possível decisão de projeto em torno dos seus resultados. Além do sua natureza avaliativa, o BQSNEKO encontra um viés pedagógico no estudo de mecanismos de tolerância a faltas bizantinas e das propriedades de registradores de leitura e escrita. Para *download* do BQSNEKO e maiores informações sobre o projeto, visite a página em <http://www.das.ufsc.br/~wagners/bqsneko>.

Referências

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

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

Goodson, G. R., Wylie, J. J., Ganger, G. R., and Reiter, M. K. (2004). Efficient byzantine-tolerant erasure-coded storage. In *DSN '04: Proceedings of the 2004 International*

Conference on Dependable Systems and Networks, page 135, Washington, DC, USA. IEEE Computer Society.

Herlihy, M. (1991). Wait-free synchronization. *ACM Transactions on Programing Languages and Systems*, 13(1):124–149.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565.

Lamport, L. (1986). On interprocess communication (part ii: algorithms). *Distributed Computing*, 1(1):203–213.

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

Liskov, B. and Rodrigues, R. (2006). Tolerating byzantine faulty clients in a quorum system. In *ICDCS '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, page 34, Washington, DC, USA. IEEE Computer Society.

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

Malkhi, D. and Reiter, M. K. (1998b). Byzantine quorum systems. *Distributed Computing*, 11(4):203–213.

Martin, J.-P., Alvisi, L., and Dahlin, M. (2002a). Minimal Byzantine storage. In *Distributed Computing, 16th international Conference, DISC 2002*, volume 2508 of *LNCS*, pages 311–325.

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

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

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

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

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

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

A Exemplo de Configuração do BQSNEKO

O quadro abaixo é um exemplo de arquivo configuração do BQSNEKO, usado como base de execução do experimento mostrado neste artigo e posterior obtenção de todos os seus resultados conforme podem ser vistos ao longo da seção 4

Como já mencionado na seção 3.4, a configuração do BQSNEKO divide-se em duas partes fundamentais:

- Configurações básicas do ambiente e da plataforma NEKO. São as linhas 4 a 15;
- Configurações particulares de BQS: protocolos de acesso (leitura e escrita) e contexto de faltas nos servidores e nos clientes. Adicionam-se aqui parâmetros de execução do cliente, tais como a quantidade de execuções de um determinado processo e o tipo de acesso que este cliente terá no sistema de quóruns. São as linhas 18 a 25).

Quadro 1. Configuração de exemplo do BQSNeko

```
1  #### arquivo de exemplo de configuração do BQSNeko ####
2
3  ## 1a parte: configurações do Neko ##
4  simulation = true
5  process.num = 6
6
7  process.initializer = lse.neko.applications.bqs.BQSInitializer
8  network = lse.neko.networks.sim.MetricNetwork
9  network.lambda = 10
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
20 # usando o número de servidores bizantinos como todo o limite de faltas
21 faulty.server.QPoisonousProfileLayer.percent = 100
22
23 qinfo = lse.neko.applications.bqs.sym.faulty.SymmFaultyMWMRSafeInfo
24 qclientlayer = lse.neko.applications.bqs.sym.faulty.SymmFaultyMWMRSafeClient
25 qserverlayer = lse.neko.applications.bqs.sym.faulty.SymmFaultyMWMRSafeServer
```

Na primeira parte da configuração, percebe-se que o experimento a ser efetuado trata-se de uma simulação (linha 4) com 6 processos no total (5 servidores, o correspondente a $4f+1$ servidores no sistema quando $f=1$, e 1 cliente) em um ambiente de rede local segundo o modelo de rede com noção de contenção explicado na seção 4, implementado pelo NEKO (linhas 9 e 10).

A segunda parte define os parâmetros do limite de faltas no sistema ($t=1$, linha 18), a quantidade f de servidores bizantinos de fato à luz do limite de faltas t ($f=t$, linha 21), o perfil de falta bizantina a ser injetado nos servidores faltosos (perfil "venenoso"); especificam-se além disso a estrutura do sistema de quóruns (objeto de dados BQSNEKO) de onde se extrai o número de 5 servidores no sistema (linha 23) e o tamanho dos quóruns de leitura e escrita, os algoritmos de acesso ao sistema na parte cliente (linha 24) e de execução do servidor (linha 25). Observe que os valores do tipo de operação do cliente (parâmetro layer.application.type.6) e do número de execuções do mesmo (parâmetro layer.application.executions.write) estão omitidos por serem empregados os seus valores padrão, respectivamente *write* e 1.