UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática


# CONCRETIZAÇÃO DE UMA BIBLIOTECA PARA REPLICAÇÃO TOLERANTE A INTRUSÕES

**André Filipe Farias de Sousa**

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Arquitectura, Sistemas e Redes de Computadores

2009

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática


# CONCRETIZAÇÃO DE UMA BIBLIOTECA PARA REPLICAÇÃO TOLERANTE A INTRUSÕES

**André Filipe Farias de Sousa**

PROJECTO

Projecto orientado pelo Prof. Doutor Alysson Neves Bessani

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Arquitectura, Sistemas e Redes de Computadores

2009

# Agradecimentos

Em primeiro, gostaria de agradecer ao Professor Doutor Alysson Neves Bessani, meu coordenador, por ter acreditado no meu trabalho e estar sempre disponível para esclarecer e discutir ideias.

Agradeço a todos os meus colegas de investigação que, ao longo do ano, proporcionaram um ambiente de trabalho agradável e de companheirismo.

Agradeço a todos os meus amigos por todos os bons momentos que passamos juntos.

Por fim, quero deixar um agradecimento especial à minha irmã pela sua amizade e aos meus pais pelo seu apoio e afecto tornando possível a realização dos meus estudos.

# Resumo

A tolerância a intrusões tem sido uma área bastante activa nos últimos anos, em parte, devido ao crescente número de ataques bem sucedidos que se têm registado. Têm surgido várias propostas para sistemas tolerantes a intrusões, sendo que, a maioria desses sistemas optam por escolher uma de duas abordagens: máquina de estados replicada ou sistemas de quórum bizantinos. Enquanto que a primeira permite realizar qualquer tipo de operação generalista, os sistemas de quoruns bizantinos conseguem actuar em ambientes completamente assíncronos mas só suportam operações de leitura e escrita. Nesta tese é apresentada uma biblioteca de replicação tolerante a intrusões denominada Objectos de Serviço Confiávies (OSC). Esta biblioteca possibilita a construção de aplicações seguras e confiáveis que manipulem objectos replicados. A OSC é baseada no Sistemas de Quoruns Activos (SQA), um modelo de replicação tolerante a faltas bizantinas já existente, sendo um dos primeiros a juntar as duas principais abordagens para suportar operações de diferentes semânticas. O SQA fornece três tipos de operações distintas: leitura, escrita e actualização. Adicionalmente, o SQA demonstra uma característica única pois permite a construção de sistemas não deterministas. Esta tese apresenta o modelo SQA e a concretização da biblioteca OSC, sendo que, a OSC foi desenvolvida de modo a fornecer uma interface que permita manipular objectos através dos protocolos do SQA. Esses objectos estão preparados para ambientes de grande concorrência apresentando uma elevada disponibilidade. Uma das particularidades mais fortes da OSC é o suporte a *multithreading*, de modo a aproveitar as arquitecturas recentes que apresentam diversos núcleos de processamento. Os testes de desempenho efectuados à biblioteca OSC permitiram obter resultados bastante interessantes e promissores relativamente a outros modelos de replicação tolerantes a intrusões. Adicionalmente foi concretizado e avaliado um serviço LDAP sobre a OSC, de forma a se comprovar que o modelo proposto pelo SQA pode ser usado para o desenvolvimento de aplicações genéricas seguras e confiáveis.

**Palavras-chave:** Segurança, Confiabilidade, Replicação, Tolerância a Faltas Bizantinas

# Abstract

Intrusion tolerance has been a very active area in recent years, in part, due to the increasing number of successful attacks that have been recorded. Several proposals have emerged for intrusion-tolerant systems, and, most these systems choose one of two approaches: state machine replication or Byzantine quorum systems. While the first allows the implementation of any type of operation, Byzantine quorum systems can operate in environments completely asynchronous but only support simple read and write operations. This thesis presents an intrusion-tolerant library for object replication called Dependable Service Objects (OSC). This library allows the construction of secure and reliable applications that manipulate replicated objects. The OSC is based on Active Quorum Systems (SQA), a Byzantine fault-tolerant replication model, which is one of the first to join the two main approaches to support operations of various semantics. SQA provides three different types of operations: read, write and read-modify-write. Additionally, SQA presents an unique feature that allows the construction of non-deterministic systems. This thesis presents the implementation of OSC that has been developed to provide a interface that allows manipulation of objects through the SQA protocols. One of the most strongest features supported by OSC is a multi-threading model in order to take advantage of recent architectures showing various processing cores. The OSC was tested and presents results quite interesting and promising compared to other types of intrusion-tolerant replication. Additionally, an LDAP service was build and evaluated over the OSC library in order to prove that the model proposed by SQA can be used to develop reliable and safe intrusion-tolerant services.

**Keywords:** Security, Dependability, Replication, Byzantine Fault Tolerance

# Conteúdo

**Lista de Figuras** **xiii**

**Lista de Tabelas** **xv**

**1 Introdução** **1**
- 1.1 Motivação . . . 2
- 1.2 Objectivos . . . 3
- 1.3 Metodologia . . . 3
- 1.4 Contribuições . . . 4
- 1.5 Organização do Documento . . . 4

**2 Tolerância a Intrusões** **5**
- 2.1 Aproximação ao tema . . . 5
- 2.2 Máquina de estados replicada . . . 7
- 2.3 Sistemas de quoruns bizantinos . . . 8
- 2.4 Máquina de estados replicada VS Sistemas de quoruns . . . 9
- 2.5 Trabalho Relacionado . . . 9
  - 2.5.1 Castro-Liskov BFT . . . 9
  - 2.5.2 Query/Update . . . 11
  - 2.5.3 BFT-BC . . . 11
  - 2.5.4 HQ: Replication . . . 12
  - 2.5.5 Zyzzyva . . . 13
- 2.6 Outras Abordagens . . . 13
  - 2.6.1 RITAS . . . 13
  - 2.6.2 Máquina de estados replicada baseada em serviços TTCBs . . . 14
- 2.7 Considerações Finais . . . 14

**3 Sistemas de Quoruns Activos** **15**
- 3.1 Modelo . . . 15
- 3.2 Descrição Geral . . . 16
- 3.3 Protocolos . . . 18
  - 3.3.1 Certificados . . . 18

3.3.2 Escrita . . . 18
3.3.3 Leitura . . . 21
3.3.4 Actualização . . . 22
3.4 Avaliação dos protocolos . . . 26
3.5 Considerações Finais . . . 27

**4 Concretização da biblioteca Objectos de Serviço Confiáveis 29**
4.1 Considerações gerais . . . 29
4.2 Java Byzantine Paxos . . . 30
4.3 Arquitectura . . . 30
4.3.1 Cliente . . . 31
4.3.2 Servidor . . . 31
4.4 Optimizações . . . 33
4.4.1 Batching de mensagens . . . 33
4.4.2 Sínteses de mensagens PAXOS (Weak, Strong, Decide) . . . 34
4.4.3 Vectores de MACS em vez de assinaturas . . . 34
4.4.4 Actualizações sem enviar estado . . . 35
4.5 Suporte a *multithreading* . . . 36
4.6 Operações em Múltiplos Objectos . . . 38
4.7 Considerações Finais . . . 38

**5 Avaliação Experimental da Biblioteca OSC 39**
5.1 Ambiente . . . 39
5.2 Latência . . . 39
5.3 Throughput . . . 41
5.3.1 Efeito dos *Multi-Cores* . . . 43
5.4 Testes ao *Communication System* . . . 44
5.5 Considerações Finais . . . 45

**6 BFT LDAP 47**
6.1 Descrição . . . 47
6.2 Operações . . . 48
6.2.1 Bind / Unbind . . . 48
6.2.2 Search . . . 49
6.2.3 Add . . . 49
6.2.4 Remove . . . 49
6.2.5 Compare . . . 50
6.2.6 Modify . . . 50
6.2.7 ModifyDN . . . 50
6.3 Integração . . . 50

6.3.1 Cliente . . . . . 50
6.3.2 Servidor . . . . . 51
6.4 Avaliação . . . . . 52
6.5 Considerações Finais . . . . . 53

**7 Conclusão** **55**
7.1 Trabalho Futuro . . . . . 56

**Bibliografia** **61**

# Lista de Figuras


2.1 Modelo AVI. . . . . 6
2.2 Protocolo CL-BFT. . . . . 10

3.1 Protocolo de escrita. . . . . 20
3.2 Protocolo de leitura. . . . . 22
3.3 Protocolo de actualização. . . . . 24
3.4 Protocolo de actualização optimizado. . . . . 26

4.1 Arquitectura do cliente e do servidor da biblioteca OSC. . . . . 30
4.2 Modelo de objectos OSC com duas classes exemplos de objectos que estendem a classe `AQSObject`. . . . . 32
4.3 Modelo de *Thread Polling* usado nos protocolos de escrita e leitura. . . . 37
4.4 Modelo de *Thread Polling* usado no protocolo de actualização. . . . . 37

5.1 Latência dos vários protocolos pela biblioteca OSC para objectos de tamanho entre 4 bytes e 16 KBs. . . . . 40
5.2 Throughput máximo dos protocolos de escrita e leitura. . . . . 42
5.3 Throughput máximo do protocolo de actualização com diferentes tamanhos de batches de mensagens. . . . . 42
5.4 Latência do *Communication System* esperando por n - f respostas. . . . . 44
5.5 Latência do *Communication System* esperando por uma resposta. . . . . 45

6.1 Modelo usado para a concretização de aplicações sobre a OSC. . . . . 47
6.2 Exemplo da estrutura de um directório LDAP. . . . . 48
6.3 Modelo de classes do LDAP. . . . . 52
6.4 Latência das operações *add* e *modify*. . . . . 53
6.5 Latência da operação *search* com diferentes números de objectos de 4KB retornados. . . . . 53

# Lista de Tabelas


3.1 Resumo dos protocolos de escrita, leitura e actualização sobre o número de passos necessários no melhor e pior caso, complexidade de mensagens e necessidade de assumpções temporais. . . . . 26

4.1 Operações do SQA e métodos do OSC correspondentes. . . . . 31
4.2 Métodos de terminação dos protocolos. . . . . 31
4.3 Tamanho das mensagens de *Propose*, em bytes, em comparação com as mensagens do PAXOS (Weak, Strong e Deceide) para objectos de 4 bytes, variando entre 1 e 3 pedidos para objectos diferentes. . . . . 34
4.4 Tempo necessário em milissegundos para efectuar assinaturas e vectores de MACs para objectos de 172 bytes. . . . . 35

5.1 Número de operações por segundo verificado para objectos de 4 bytes utilizando uma máquina com um *core* e outra com *quad-core*. Para a actualização foi usado um *batching* máximo de 200 mensagens. . . . . 43

6.1 Operações concretizadas pelo LDAP e respectivos protocolos SQA. . . . . 51

# Capítulo 1

# Introdução

A necessidade de serviços informáticos confiáveis e altamente disponíveis tem aumentado consideravelmente nos últimos anos. Este facto deve-se, em grande parte, ao elevado número de vulnerabilidades e incidentes de segurança (ataques) que têm sido notificados ano após ano. Como consequência, o paradigma da tolerância a intrusões [11, 23] tem sido uma área de investigação bastante activa e promissora tendo juntado áreas como a segurança e confiabilidade. Este conceito de tolerância a intrusões permite o desenvolvimento de sistemas que, mesmo na eventualidade de certas componentes serem comprometidas por um atacante, continuam a fornecer o seu serviço conforme especificado.

Grande parte dos sistemas relacionados com a tolerância a intrusões, têm como foco um certo tipo de faltas mais generalista denominada de tolerância a faltas bizantinas (BFT) [15]. Faltas arbitrárias, comportamentos estranhos por parte da rede ou até mesmo ataques maliciosos a componentes do sistema são exemplos de tipos de faltas que os sistemas BFT's admitem existir, sendo que, para a sua concepção, tipicamente, são usadas dois tipos de técnicas: máquina de estados replicada e sistemas de quoruns. Ao longo desta tese, este tema irá ser abordado assim como o estudo do estado da arte para protocolos tolerantes a faltas bizantinas.

Para além dessa discussão, irá ser apresentado uma novo protocolo da família BFT que tem a particularidade de juntar sistemas de quoruns com máquinas de estados replicadas para, deste modo, possibilitar o desenvolvimento de operações com diferentes semânticas. Esta nova proposta chama-se Sistemas de Quoruns Activos (SQA), conforme definido em [4], sendo este um modelo de replicação no qual, o estado global do sistema é definido por um conjunto de objectos de dimensões reduzidas, replicados nos diferentes servidores que compõem o sistema. Sobre estes objectos é possível realizar três tipos de operações: escrita, leitura e actualização (também conhecido como *read-modify-write*).

O facto do SQA dividir o estado do sistema pelos diferentes objectos e as operações em classes diferentes oferece dois benefícios básicos: (1.) promove uma maior separação entre as partes distintas do estado do serviço (e.g., num serviço NFS, operações que actualizam dois ficheiros distintos não precisam de ser sincronizadas); (2.) cada uma das

classes de operações podem ser concretizadas pelos protocolos mais simples disponíveis (em termos de hipóteses requeridas do ambiente e complexidade de mensagens).

Esta tese descreve o SQA e os seus protocolos assim como a biblioteca Objectos de Serviço Confiáveis (OSC) que concretiza este modelo de replicação. Esta biblioteca foi desenvolvida de modo a aumentar ao máximo a disponibilidade apresentada pelos vários objectos dos sistema. Para isso, foram aplicadas técnicas como o suporte a *multithreading* e o controle de concorrência através de *locks* de leitura/escrita. Foram realizadas vários experimentos com a biblioteca OSC de forma a ser possível realizar uma análise critica ao seu desempenho.

Adicionalmente, é apresentado um LDAP tolerante a faltas bizantinas desenvolvido sobre a biblioteca OSC assim como a sua respectiva avaliação.

## 1.1 Motivação

A sociedade actual está cada vez mais dependente de sistemas informáticos, geralmente interligados entre si. A evolução da Internet nos últimos anos, assim como as facilidades de acesso, têm contribuido para um grande crescimento da procura e da oferta de serviços *online*. Organizações governamentais e militares, empresas, ou até mesmo utilizadores comuns estão dependentes desse tipo de serviços, sendo que, por vezes, até serviços mais críticos (ex: gás, electricidade) necessitam de infraestruturas ligadas à Internet.

A par desta evolução, surgiu o paradigma da segurança informática que preocupa-se com aspectos como a confidencialidade, integridade, autenticidade e disponibilidade e, para isso, foram investigadas e desenvolvidas inúmeras técnicas para auxiliar a segurança dos sistemas. Algoritmos de criptografia, mecanismos de controlo de acesso e anteparas de segurança são apenas alguns exemplos dessas técnicas. No entanto, como comprovado historicamente, esses mecanismos de segurança convencionais já não são suficientes para lidar com o número e tipo de ameaças que têm surgido. O elevado número de ataques com sucesso reportados demonstram esse facto. Estes modelos tradicionais referenciados, tipicamente, têm uma abordagem de prevenção de ataques, de modo a impedir que estes aconteçam. No entanto, tendo em conta a elevada dimensão e complexidade que certos sistemas atingem, torna-se bastante difícil prever todos os passos de um atacante, sendo que, por vezes, este consegue contornar os mecanismos de segurança sem que o ataque seja detectado. Outro factor a ter em conta é que grande parte dos ataques actuais são originados através de engenharia social (ex: atacante consegue obter palavra-chave de um utilizador induzindo-o em erro) e, como consequência, o atacante poderá parecer um utilizador legitimo, ter acesso a parte do sistema e possibilidade de comprometer certas componentes vitais. Com base nestes factos, tem sido investigado o paradigma da tolerância a intrusões [11, 23] (uma sub-classe de tolerância a faltas bizantinas) que permite que o sistema se comporte correctamente, mesmo na presença de uma falta causada

por uma intrusão (detectada ou não), continuando a fornecer o serviço de forma segura.

A opção de construir este tipo de sistemas requer uma análise crítica ao ambiente onde vão ser executados, o nível de segurança que se pretende atingir e também a que tipo de ameaças vão estar sujeitos. Os sistemas tolerantes a intrusões recorrem a algoritmos de replicação, o que quase sempre acarreta numa perda de desempenho do sistema. Este aspecto é bastante importante pois requer que os protocolos utilizados sejam desenvolvidos de modo a, de alguma forma, aliviar o impacto que o uso dos algoritmos de replicação têm no sistema. Nos últimos anos, esta área de investigação tem sido alvo de curiosidade de diversos grupos de investigação, tendo surgido diversos modelos de replicação tolerantes a faltas bizantinas. A proposta de Castro e Liskov (CL-BFT) [5] foi bastante importante para motivar a construção deste tipo de sistemas pois provou que é possível desenvolve--los mantendo um desempenho bastante aceitável, tornando-se uma abordagem que pode ser seguida na concretização de serviços confiáveis.

De modo a comprovar estes factos descritos, nesta tese, é desenvolvido um sistema tolerante a faltas bizantinas baseado no modelo Sistemas de Quoruns Activos [4] com o objectivo de se concretizar uma biblioteca que permita a construção de serviços confiáveis e seguros em ambientes bizantinos.

## 1.2 Objectivos

O principal objectivo deste projecto é o desenvolvimento de uma biblioteca tolerante a intrusões usando como especificação a proposta de Sistemas de Quoruns Activos [4]. Essa biblioteca irá conter protocolos que concretizem as operações permitidas pelo SQA e pretende-se que seja encarada como uma camada entre o ambiente e a aplicação, devendo o processo de integração ser o mais simples possível.

Pretende-se realizar diversos tipos de testes ao sistema, analisando o seu desempenho e comportamento de modo a, se necessário, desenvolver certas optimizações aos protocolos originais nos passos que se revelem mais penalizantes.

Outro dos objectivos deste projecto é a concretização de uma aplicação exemplo que utilize a biblioteca do SQA de modo a demonstrar a sua utilidade e usabilidade. O tipo de serviço escolhido foi um LDAP simplificado no qual as operações suportadas são realizadas recorrendo aos protocolos do SQA.

## 1.3 Metodologia

Este projecto foi desenvolvido na unidade de investigação *LaSIGE*[1], situado no Departamento de Informática na Faculdade de Ciências da Universidade de Lisboa. Fui inserido no grupo de investigação *Navigators*, no qual o meu orientador de projecto, o Professor

[1] `http://lasige.di.fc.ul.pt`

Doutor Alysson Neves Bessani, também está inserido. Por este motivo de proximidade, ao longo do projecto foi possível a existência de uma boa comunicação de forma a que certas ideias que tenham surgido fossem rapidamente discutidas.

No inicio deste projecto, foi necessário realizar um estudo acerca do tema tolerância a intrusões de forma a adquirir conhecimentos nesta área de investigação. Após este processo foi iniciado a concretização do Sistemas de Quoruns Activos. De notar que a especificação deste modelo foi proposto pelo meu orientador. Esta fase de implementação foi realizada de forma incremental tendo sido efectuadas várias revisões de modo a melhorar o sistema. À medida que se efectuava uma nova versão, eram realizados testes ao desempenho de modo a se perceber o que poderia ser melhorado. Foi através deste processo, que muitas das optimizações apresentadas no capítulo 4, foram realizadas.

## 1.4 Contribuições

A principal contribuição desta tese é a concretização dos algoritmos para o Sistemas de Quoruns Activos (SQA) [4] de modo a disponibilizar uma biblioteca tolerante a intrusões denominada Objectos de Serviço Confiáveis (OSC). Essa biblioteca fornece operações que permitem a manipulação de objectos generalistas replicados, sendo que, durante o seu desenvolvimento, tomaram-se opções a pensar em propriedades como a escalabilidade e disponibilidade.

Outra contribuição importante foi a analise experimental efectuada à biblioteca concretizada. Os resultados obtidos permitem efectuar uma avaliação aos protocolos do SQA.

## 1.5 Organização do Documento

Esta tese está organizada da seguinte forma: no capítulo 2 é introduzido o conceito de tolerância a faltas bizantinas, técnicas usadas assim como um estudo realizado acerca das propostas de sistemas BFTs mais conceituadas; o capítulo 3 apresenta o Sistema de Quoruns Activos, suas componentes e e suas principais características; o capítulo 4 é referente à concretização do SQA resultando na biblioteca Objectos de Serviço Confiáveis (OSC), as técnicas usadas e as optimizações efectuadas. O capítulo 5 apresenta a avaliação experimental do SQA assim como uma discussão acerca dos seus resultados. Por fim, o capítulo 6 é referente ao desenvolvimento de um caso pratico de uma aplicação concretizada sobre o SQA e no capítulo 7 estão apresentadas as conclusões do projecto, assim como os possíveis trabalhos futuros.

# Capítulo 2

# Tolerância a Intrusões

## 2.1 Aproximação ao tema

A utilização crescente dos sistemas distribuídos, em várias áreas de actividade, levou a uma maior preocupação em relação à confiabilidade das diferentes componentes de um sistema [23, 7, 20]. Um dos aspectos mais importante nos modelos de sistemas distribuídos clássicos é a **tolerância a faltas** que tem o objectivo de aumentar a disponibilidade e fiabilidade dos sistemas. Ou seja, mesmo na eventualidade de ocorrer qualquer problema que possa comprometer parte do sistema, este deve ter a capacidade de continuar a fornecer o seu serviço de forma correcta. Este paradigma levou a uma mudança de pensamento em relação ao funcionamento dos sistemas distribuídos. Foi adoptado uma atitude pessimista, na qual se pensa que nenhum sistema é 100% correcto, de tal forma, que é assumido terem sido cometidos lapsos durante a especificação, desenho ou concretização do sistemas que, eventualmente, poderão dar origens a faltas.

Tipicamente, os sistemas distribuídos estão assentes neste modelo de faltas:

$$Falta- > Erro- > Falha$$

A tolerância a faltas não impede que as faltas aconteçam, no entanto, tem mecanismos próprios para evitar que essas faltas originem erros e como consequência falhas no sistema. Poderia pensar-se que seria mais óbvio prevenir essas faltas em vez de as tolerar, reduzindo desta forma o risco de falha do sistema, no entanto, e tendo em conta a complexidade de certos sistemas, torna-se bastante complicado ou até mesmo impossível prever todas as faltas possíveis visto que as suas origens podem ser causadas por diversos motivos, tanto a nível interno como a nível externo. A técnica encontrada para se conceber sistemas tolerantes a faltas é recorrer a algoritmos de replicação. Esta abordagem de distribuir os serviços, contribui para a o aumento da resiliência às faltas mais comuns pois, ao contrário de um sistema centralizado, não existe um único ponto de falha.

Este modelo tradicional de tolerância a faltas apenas tem em consideração a possibilidade de acontecerem faltas benignas, ou seja, faltas que ocorrem sem intenção maliciosa.

No paradigma da tolerância a faltas, as técnicas usadas assumem que certos componentes do sistema podem falhar por paragem ou por omitirem certos passos do protocolo, ou seja, se uma componente falhar por outra razão (causada com intencionalidade) o sistema pode não estar preparado e ter o seu serviço comprometido.

O crescente número de ataques bem sucedidos, em sistemas distribuídos, levou ao aparecimento do conceito de **tolerância a intrusões** [11, 23]. Nesta abordagem, o sistema está preparado para lidar com faltas causadas de forma maliciosa (intrusões). Normalmente, se este tipo de faltas não for tratado poderá provocar uma falha de segurança no sistema, o que irá comprometer o seu comportamento. O paradigma da tolerância a intrusões surge da junção de duas linhas de investigação distintas: segurança e confiabilidade. Deste modo, ao seguir esta abordagem, torna-se possível desenvolver sistemas que respeitem as propriedades da segurança e, ao mesmo tempo, sejam tolerantes a faltas.

A tolerância a intrusões está assente no modelo ilustrado na Figura 2.1.

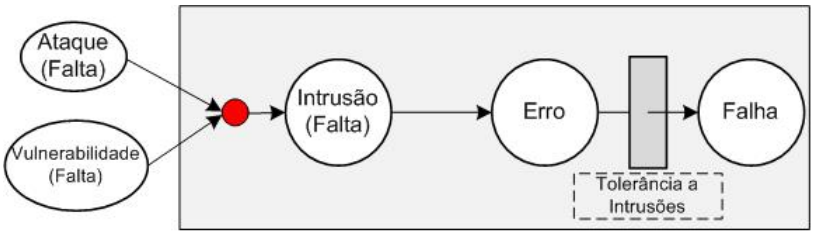


Figura 2.1: Modelo AVI.

Este modelo denomina-se por modelo AVI (Ataque, Vulnerabilidade, Intrusão) [25] e é uma extensão ao modelo tradicional de faltas. Uma intrusão, só ocorre quando um atacante consegue explorar uma certa vulnerabilidade do sistema. Essa vulnerabilidade pode ter origem a nível de projecto (e.g., falta de um controlo de acesso), codificação (e.g., *bug* pode gerar um *buffer oferflow*) ou a nível operacional (e.g., *bug* no sistema operativo). Se um atacante descobrir forma de aproveitar essa vulnerabilidade, pode dar origem a uma intrusão. Essa intrusão é considerada como uma falta que pode desencadear o aparecimento de erros e consequentemente falhas no sistema.

Tal como na tolerância a faltas clássica, a tolerância a intrusões não impede que as intrusões (faltas) aconteçam. É assumido que as aplicações têm vulnerabilidades e que as intrusões são uma realidade, podendo mesmo algumas delas causar erros e até comprometer parte do sistema. No entanto, o sistema tem a capacidade para prevenir que essas intrusões se espalhem pelo sistema e comprometam o serviço concretizado.

Outro conceito que está directamente ligado com a tolerância a intrusões, é a **tolerância a faltas bizantinas** (BFT) [15]. Faltas bizantinas são a classe de faltas mais generalista possível englobando todos os tipos de faltas arbitrárias que podem ocorrer durante a execução de um algoritmo. Erros no software, ataques maliciosos ou até mesmo erros por parte do utilizador são exemplos de faltas bizantinas.

A maior parte dos trabalhos relacionados com a tolerância a intrusões assumem que o seu sistema é executado num ambiente bizantino, ou seja, todos os processos estão sujeitos a faltas arbitrárias, sejam elas causadas por um atacante, por uma falha de *software* ou por motivos externos ao sistema. Ao longo deste relatório o termo BFT irá ser bastante usado sendo que foi o modelo escolhido para o desenvolvimento do projecto.

Tal como na tolerância a faltas clássica, é necessário recorrer ao paradigma da replicação para desenvolver sistemas BFT's. Neste momento, existem duas abordagens principais para realizar a replicação: **máquina de estados replicada** [22] e **sistemas de quoruns bizantinos** [11]. Estas duas técnicas têm vantagens e desvantagens quando comparadas uma com a outra sendo que a escolha deve ser baseada no tipo e complexidade do serviço confiável a ser concretizado. Nas próximas secções, estas técnicas são discutidas sendo que, de seguida, são apresentadas algumas das propostas mais importantes que têm surgido nos últimos anos para a concretização de sistemas BFT ou tolerantes a intrusões.

## 2.2 Máquina de estados replicada

O paradigma da máquina de estados replicada é uma forma generalista de desenvolver serviços tolerantes a faltas em sistemas distribuídos. Os servidores são replicados, sendo que cada um representa uma máquina de estados que contêm variáveis e operações que permitem manipular essas variáveis de estado. O estado global do sistema é definido pelo conjunto de todas as variáveis. Todos os servidores começam com o mesmo estado sendo que os clientes alteram-no através da invocação de operações, de tal forma que essas operações têm de ser atómicas em todos os servidores. A máquina de estados replicada tem o principio de que cada réplica tem de manter o seu estado consistente com o resto do sistema. Para alcançar este objectivo, cada réplica executa as operações de forma a satisfazer as seguintes propriedades [22]:

**Acordo:** todos os servidores executam as mesmas operações;

**Ordem:** todos os servidores executam as operações pela mesma ordem.

A satisfação destas propriedades requer o uso de algoritmos distribuídos que ofereçam certas garantias sobre a entrega das requisições ao conjunto de réplicas. As várias operações executadas pelas réplicas têm de ser deterministas pois, o resultado destas tem de ser o mesmo nas diversas réplicas do sistema. Outra propriedade que está inerente ao uso de algoritmos de acordo (consenso) é que o sistema tem de fazer assumpções de tempo para assegurar a sua terminação, pois não é possivel resolver consenso de forma determinista em sistemas assíncronos [10].

## 2.3 Sistemas de quoruns bizantinos

Sistemas de quoruns bizantinos podem ser definidos como um conjunto de sub-conjuntos de um grupo de servidores (denominados por quoruns) tolerantes a faltas bizantinas [16]. Os quoruns apresentam duas propriedades bastante importantes: intersecção e disponibilidade. A primeira propriedade garante que as operações efectuadas nos diferentes quoruns mantêm-se consistentes enquanto que a disponibilidade é ganha pois cada quorum tem a capacidade de actuar em nome do sistema. Através de um sistema de quoruns é possível definir objectos distribuídos e sobre eles realizar operações de tal forma que, por exemplo, pode ser usado para simular a existência de memória partilhada fiável.

A principal diferença entre o tipo de operações que os sistemas de quoruns permite e os da máquina de estados replicada é o tipo de semântica suportado para as operações. Na abordagem apresentada anteriormente, é possível realizar qualquer tipo de operação independentemente da sua semântica ou complexidade. Nos sistemas de quoruns apenas se consegue concretizar operações de leitura e escrita que não requerem o conhecimento prévio do estado do objecto para serem realizadas. Apesar da simplicidade, é garantido que, através da propriedade de intersecção, se uma escrita é efectuada num quorum e, de seguida é efectuada uma operação de leitura, o valor retornado é o resultado da escrita realizada.

Outra característica vantajosa que, tipicamente, os sistemas de quoruns demonstram é a sua elevada escalabilidade. Este facto é, em parte, explicado através da propriedade de disponibilidade, que permite que as operações não necessitem de ser executados em todos os quoruns.

Tipicamente, para implementar sistemas de quoruns bizantinos são usados grupos compostos por $3f + 1$ servidores com quoruns de tamanho $2f + 1$. Desta forma é assegurado que, mesmo na eventualidade de acontecerem $f$ faltas, existem pelo menos dois quoruns que se intersectam numa réplica correcta. Isto significa que, cada dois quoruns mantêm um número de servidores correctos de tal forma que, pelo menos um quórum é formado por servidores correctos.

Normalmente, o estado de um registo em cada servidor é representado pelo seu valor e por uma estampilha temporal (*timestamp*). A operação de escrita é processada da seguinte forma: a estampilha é lida através dos quoruns, incrementada e, posteriormente, é escrito um novo valor para o sistema de quoruns juntamente com a estampilha gerada. Na leitura, o resultado retornado é um par composto pelo valor e respectiva estampilha temporal. Em alguns dos sistemas de quoruns bizantinos estudados, em que seja previsível haver concorrência entre operações, para assegurar que o valor retornado de uma leitura é o mais recente, pode ser usado um mecanismo denominado de *writeback* no qual o valor lido é escrito no sistema. Deste modo, todas as leituras realizadas posteriormente irão retornar o mesmo par valor, estampilha temporal ou um par mais recente.

## 2.4 Máquina de estados replicada VS Sistemas de quoruns

Como descrito nas secções anteriores, a máquina de estados replicada e os sistemas de quoruns (bizantinos) são duas abordagens completamente distintas que têm o objectivo de realizar a replicação de objectos em ambientes bizantinos. Esta secção compara as duas técnicas relevando as vantagens e desvantagens de cada uma delas.

Comparativamente com a máquina de estados replicada, os sistemas de quoruns bizantinos são bastante mais simples, tanto ao nível da sua construção como da complexidade de mensagens, principalmente por não requererem algoritmos de acordo. Por este motivo também, surgem mais vantagens importantes: (1.) os sistemas de quoruns são mais escaláveis do que os sistemas baseados da máquina de estados [1]; (2.) está assente num modelo assíncrono (sem assumpções em relação ao tempo). A máquina de estados replicada, por outro lado, requer algumas assumpções temporais para assegurar a terminação dos seus algoritmos. No entanto, a grande vantagem desta abordagem é a possibilidade de se concretizar operações com uma semântica mais forte do que as operações de leitura e escrita permitidas nos sistemas de quoruns bizantinos. Esta é mesmo a grande diferença entre as duas técnicas pois, com sistemas de quoruns não é possível concretizar qualquer tipo de serviço genérico.

A máquina de estados replicada é uma abordagem óptima a seguir em cenários em que exista contenção ou que se pretenda executar operações complexas (e.g., operações que necessitem do estado actual das variáveis para determinar o resultado). Caso contrário, quando não existe contenção ou as operações a processar são simples (e.g., leituras e escritas), a escolha deste modelo para a implementação de um sistema BFT pode acarretar numa perda de desempenho significativa pois, a necessidade de recorrer a algoritmos de acordo, revela-se bastante penalizadora.

Em termos do número de processos necessários estas, técnicas revelam-se equivalentes, pois, tipicamente, ambas requerem a existência de $3f + 1$ servidores, sendo $f$ o número de falhas possíveis.

## 2.5 Trabalho Relacionado

Nos últimos anos, várias têm sido as propostas para sistemas tolerantes a faltas bizantinas com resultados interessantes. Alguns dos protocolos optam por utilizar a abordagem da máquina de estados replicada enquanto outros preferem a simplicidade dos sistemas de quoruns bizantinos. Nesta secção, são apresentados alguns desses protocolos.

### 2.5.1 Castro-Liskov BFT

A proposta de Castro-Liskov BFT (CL-BFT) é um modelo de máquina de estados replicada que concretiza um PAXOS tolerante a faltas arbitrárias. É assegurado que todas as

réplicas executam as operações pela mesma ordem, através de um algoritmo de difusão de ordem total [5]. Este algoritmo faz uso de um processo líder, cuja principal função é propor uma ordem de execução para todos os pedidos, atribuindo um número de sequência a cada mensagem. Quando o líder falha, um algoritmo de troca de vista faz com que um novo processo seja eleito líder.

O protocolo concretizado pelo CL-BFT é composto por três fases: PRE-PREPARE, PREPARE e COMMIT sendo iniciado pelo cliente ao enviar um pedido $m$ para todos os servidores. O líder é responsável por propor uma ordem para os pedidos, de tal forma, que envia para todas as réplicas uma mensagem PRE-PREPARE, contendo $m$ e um número de sequência, $i$. Os servidores aceitam esta mensagem, caso a proposta do líder seja considerada boa[1] e enviam a mensagem PREPARE para todos os servidores. Após um servidor receber $(n + f)/2$ mensagens PREPARE com os mesmos valores para $m$ e $i$, marca $m$ como preparada e envia uma mensagem do tipo COMMIT com $m$ e $i$ para todos os servidores. Quando um servidor recebe $(n + f)/2$ mensagens COMMIT aceita a mensagem m que será executada quando todas as mensagens com números de sequência inferiores a $i$ forem executadas. Cada servidor, ao processar o pedido $m$, envia a resposta para o cliente em questão, que espera por $n - f$ respostas para retornar o resultado recebido. A fase PREPARE do protocolo assegura que não existem duas mensagens com o mesmo número de sequência enquanto que a fase COMMIT assegura que a mensagem $m$ irá ser executada com o número de sequência proposto mesmo que o líder seja faltoso. Este procedimento é ilustrado na Figura 2.2.

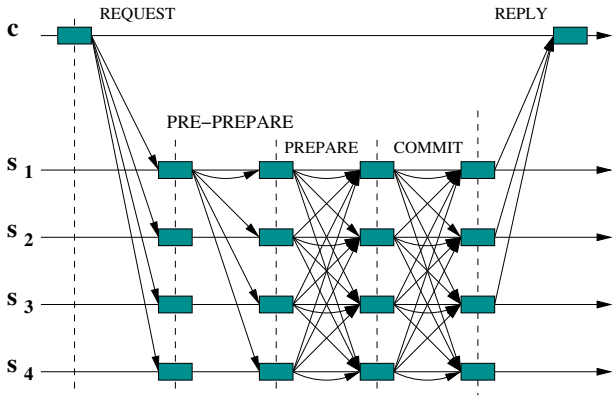


Figura 2.2: Protocolo CL-BFT.

Quando o líder é suspeito de ser faltoso é iniciado um processo de eleição no qual, cada servidor envia o seu estado para o novo líder que, deste modo, tem a informação necessária para assegurar o comportamento correcto do protocolo. Através da troca de estados entre as réplicas e dos números de sequência de mensagens pendentes é possível que os vários servidores obtenham um novo estado consistente.

[1] O CL-BFT tem mecanismos próprios para verificar a validade das mensagens e, adicionalmente, é verificado se o número de sequência proposto está dentro de um certo intervalo.

Três características interessantes do CL-BFT são: (1.) a utilização de vectores de MAC em vez de assinaturas de chave pública para verificar a autenticidade das mensagens, tornando o protocolo bastante mais eficiente; (2.) o uso de certificados (conjuntos de mensagens válidas) para possibilitar a verificação da fiabilidade da informação armazenada e (3.) a formação de *batches* de mensagens que permite agrupar conjuntos de mensagens de modo a serem ordenadas de uma só vez.

Um dos principais contributos deste protocolo foi ser o primeiro a demonstrar que é possível construir sistemas tolerantes a faltas bizantinas sem prejudicar gravemente o desempenho do sistema, sendo que serviu como uma espécie de impulsionador nesta linha de investigação.

### 2.5.2 Query/Update

O protocolo *Query/Update* (Q/U) suporta a concretização de serviços arbitrários deterministas [1] em ambientes assíncronos, sendo que é baseado em sistemas de quoruns bizantinos. Esta proposta introduz uma propriedade interessante em protocolos BFT: escalabilidade de faltas (*fault-scalability*), que permite que o aumento da quantidade de faltas toleradas pelo sistema não diminua gravemente o seu desempenho.

Este modelo permite fornecer serviços similares aos fornecidos pela técnica máquina de estados replicada sendo possível desenvolver objectos que suportem dois tipos de operações: leituras (*queries*), que não alteram o estado do objecto, e actualizações (*updates*), que alteram esse estado. Por usar sistemas de quoruns as operações, quando não existe contenção, apresentam uma complexidade de mensagens linear e latência bastante baixa (um acesso ao sistema).

Um dos pontos fracos do protocolo é que, ao contrário dos protocolos apresentados anteriormente, necessita de $5f+1$ servidores para funcionar correctamente. Este facto, pode pesar bastante na escolha deste protocolo, para o desenvolvimento de serviços visto haver necessidade de aumentar o número de máquinas físicas e também por aumentar o número de pontos de falhas possíveis no sistema. Por outro lado, demonstra uma eficiência elevada mesmo com um número de faltas grandes, o que não acontece com a maioria de protocolos BFT.

### 2.5.3 BFT-BC

O conceito original de sistemas de quoruns bizantinos previa apenas que o sistema se comportava correctamente assumindo faltas arbitrárias das réplicas, ou seja, não estava preparado para lidar com clientes bizantinos. Este protocolo apresenta uma solução em que é esperado a existência de clientes bizantinos, ou seja, clientes correctos deverão conseguir realizar operações independentemente da existência de clientes maliciosos [16].

O BFT-BC suporta operações de escrita e leitura e, como em outros sistemas de

quoruns bizantinos, para realizar as operações utiliza estampilhas temporais lógicas. A escrita é processada em dois ou três acessos ao sistema[2] (dependendo da existência de contenção) sendo que, num cenário normal, a primeira fase serve para a obtenção da estampilha temporal (fase *READ-TS*) e de seguida realiza a escrita (fase *WRITE*). Caso a primeira fase resulte em estampilhas diferentes (este cenário é possível caso haja concorrência) é necessário realizar uma fase adicional (*PREPARE*) de modo a que a escrita seja realizada com a estampilha mais recente. Relativamente á operação de leitura, apenas é necessário executar um acesso ao sistema para ser concluida. O cliente envia o pedido e espera por um quorum válido para terminar. Excepcionalmente, a leitura pode ser executada em dois acessos pois, se no primeiro passo os servidores retornarem diferentes estampilhas, é necessário proceder ao mecanismo de *writeback*.

Tal como no CL-BFT [5], este protocolo também utiliza o conceito de certificado, de tal forma que, podem ser formados dois tipos distintos de certificados: *prepare certificate* e *write certificate*. O primeiro permite que o pedido de escrita do cliente seja inicialmente aprovado por um quorum de servidores, enquanto que o segundo é formado no fim da operação permitindo desta forma validar a escrita.

A principal contribuição deste protocolo é a de fornecer um modelo tolerante a faltas bizantinas com um desempenho bastante aceitável num ambiente assíncrono que impede os clientes bizantinos de interferir com os clientes correctos.

### 2.5.4 HQ: Replication

O protocolo *Hybrid Quorum Replication* é uma proposta que utiliza sistemas de quoruns para executar as suas operações quando não existe contenção. Caso exista, o protocolo transforma-se num modelo de máquina de estados replicada tendo capacidade para ordenar as diversas operações [9]. Este trabalho surgiu da análise do protocolo Q/U [1] que, na presença de contenção, revela resultados bastante baixos.

É possível realizar operações de escrita e leitura sendo que, em cenários sem concorrência, as operações são realizadas em dois e quatro passos respectivamente. Por outro lado, caso seja detectada concorrência, é necessário recorrer a algoritmos de consenso o que introduz bastantes mais passos tanto na leitura como na escrita.

Esta proposta é bastante interessante pois consegue juntar os dois tipos de abordagens para sistemas BFTs num único protocolo de modo a escolher que tipo de modelo utiliza consoante a existência ou não de contenção.

[2]Considera-de que um acesso é composto pelo envio de uma mensagem por um cliente e da espera pela respectiva resposta.

### 2.5.5 Zyzzyva

Este protocolo utiliza a especulação para optimizar os sistemas BFT baseados na máquina de estados replicada [14]. Na prática isto significa que, em vez de executar um protocolo de três fases de *commit* para a ordenação dos pedidos, os servidores enviam directamente o resultado para o cliente após receberem a mensagem que define a ordem do líder. No Zyzzyva existe um líder que propõe a ordem em apenas duas fases e, como consequência, não é assegurado que todas as réplicas executem as operações pela mesma ordem. As réplicas respondem aos pedidos dos clientes sem executarem nenhum algoritmo que assegure a consistência do sistema, de tal forma que, essas respostas são consideradas especulativas. Por este motivo, é possível que, temporariamente, as réplicas encontrem-se em estados inconsistentes mas, este facto não impede o progresso do sistema. Neste protocolo, o cliente assume um papel bastante importante pois é ele que tem a capacidade de detectar as possíveis inconsistências e de ajudar as diferentes réplicas a convergirem para um só estado consistente. Isso é possível pois, juntamente com o resultado da operação, os servidores enviam um histórico que permite que os clientes verifiquem se os estados são consistentes. Esta proposta revelou-se bastante eficiente e inovadora tendo em conta os outros sistemas BFT existentes e, permite reduzir o número de passos necessário para realizar algoritmos de acordo.

## 2.6 Outras Abordagens

Anteriormente, foi referido e discutido as duas principais formas de desenvolver sistemas tolerantes a faltas bizantinas. No entanto, existem outras alternativas, tendo sido lançadas algumas propostas. Nesta secção são referidos alguns desses trabalhos.

### 2.6.1 RITAS

O uso de algoritmos aleatórios para a resolução do consenso permite que estes sejam processados em ambientes assíncronos, sendo que a sua terminação é assegurada de forma probabilista. A grande desvantagem desta abordagem é que, tipicamente, para os mesmos protocolos, é necessário executar bastantes mais passos de comunicação aumentado assim a complexidade sendo que o desempenho geral do sistema é gravemente afectado.

Recentemente, foi proposto o RITAS (*Randomized Intrusion-Tolerant Asynchronous Services*) [19] que explora a viabilidade deste tipo de algoritmos para construir serviços tolerantes a intrusões. Este modelo aposta na composição de várias camadas sendo que, cada uma delas, concretiza uma certa parte do sistema essencial para realizar a difusão, ordem e acordo.

O principal contributo deste trabalho foi demonstrar a possibilidade de se desenvolver serviços tolerantes a intrusões usando algoritmos aleatórios sem que o seu desempenho

seja necessariamente ruim.

### 2.6.2 Máquina de estados replicada baseada em serviços TTCBs

Este trabalho, apesar de ser baseado no paradigma da máquina de estados replicada, apresenta-se como uma alternativa interessante já que oferece duas grandes vantagens: o sistema pode operar num modelo assíncrono e pode ser composto por apenas $2f + 1$ réplicas (ao contrário dos $3f + 1$ demonstrados pela maior parte dos trabalhos) [8]. Esta proposta consegue obter estas vantagens através do recurso a TTCBs (*Trusted Timely Computing Base*), que são componentes seguras e resistentes a qualquer ataque. Estas componentes operam em tempo real oferecendo um conjunto de serviços que não podem ser afectados por faltas maliciosas, como por exemplo, a definição da ordem para os vários pedidos. Cada réplica tem um TTCB local que é responsável pela comunicação entre os servidores, sendo esta efectuada num canal de comunicação separado dos clientes.

Uma evolução mais recente deste trabalho é apresentada em [24], onde a replicação com máquina de estados é efectuada com $2f + 1$ processos através da utilização de mecanismos confiáveis locais que não necessitam de estar ligados entre si.

## 2.7 Considerações Finais

Neste capítulo foram discutidos os paradigmas da tolerância a faltas e tolerância a intrusões, convergindo estas para um modelo mais generalista de tolerância a faltas bizantinas e, o porquê de ser necessário adoptar este tipo de abordagens na construção de aplicações seguras e confiáveis.

Foram estudadas as duas técnicas mais utilizadas para se concretizar sistemas tolerantes a faltas bizantinas: (1.) máquina de estados replicada e (2.) sistemas de quoruns bizantinos; assim como algumas das propostas de sistemas que têm surgido nos últimos anos utilizando estas técnicas.

Este estudo sobre o estado da arte foi importante pois serviu de base para a percepção de um nova proposta de um modelo de replicação tolerante a faltas bizantinas denominada Sistemas de Quoruns Activos [4], que irá ser apresentado no próximo capítulo.

# Capítulo 3

# Sistemas de Quoruns Activos

Este capítulo apresenta uma nova proposta de um sistema de replicação de objectos tolerante a faltas bizantinas denominada de Sistemas de Quoruns Activos (SQA) [4]. Como irá ser descrito, o SQA tem características únicas, relativamente a outros modelos BFTs conceituados. Ao longo do capítulo, irá ser descrito a forma como o SQA funciona, o seu modelo e protocolos que concretiza.

## 3.1 Modelo

O sistema é composto por um grupo de servidores $S = \{s_0, ... , s_{n-1}\}$ e por um número arbitrário de clientes. Tanto os servidores como os clientes podem falhar de forma bizantina sendo de esperar comportamentos que não correspondem à especificação do sistema. Relativamente aos servidores, assume-se independência de falhas, ou seja, a probabilidade de um servidor falhar não é afectada pela existência de outros servidores faltosos no sistema. Os clientes correctos deverão conseguir realizar operações independentemente da existência de outros clientes bizantinos. Servidores e clientes podem ser considerados correctos ou faltosos. São considerados faltosos quando o seu comportamento não está de acordo com a sua especificação.

Em termos temporais, o sistema requer sincronia parcial para operar correctamente pois, dos três protocolos disponibilizados pelo SQA, um deles requer sincronismo para assegurar a terminação.

A comunicação é efectuada através de canais ponto a ponto fiáveis e autenticados. Para a autenticação é utilizado MACs (Message Authentication Codes). É assumida a presença de uma infraestrutura de chave pública de modo a fornecer, a cada processo, a chave pública dos restantes processos do sistema. Os pares de chaves (publica e privada) são usadas para assinar e consequentemente verificar a validade das mensagens. Na descrição dos protocolos irá ser usado o símbolo $\sigma$ para representar uma mensagem assinada. Adicionalmente, é assumida a existência de uma função de síntese, H, resistente a colisões.

## 3.2 Descrição Geral

Nos últimos anos muitas têm sido as propostas para sistemas tolerantes a faltas bizantinas. Tanto a abordagem da máquina de estados replicada como a abordagem dos sistemas de quoruns bizantino já provaram que são soluções válidas para a concretização deste tipo de sistema.

O modelo Sistemas de Quoruns Activos (SQA) surge da constatação de que, apesar dos sistemas de quoruns apresentarem uma escalabilidade e simplicidade maior que os protocolos de máquinas de estados, apenas podem ser desenvolvidos para concretizar operações simples. Operações mais complexas que necessitem de envolver algum tipo de acordo entre servidores têm de ser concretizadas recorrendo à máquinas de estados replicada. O SQA é uma proposta que junta as vantagens das duas abordagens num único sistema, tendo este a capacidade de se adaptar consoante o tipo de operação a realizar. Utiliza sistemas de quoruns bizantinos para concretizar operações de escrita e leitura e máquina de estados baseada na proposta de CL-BFT para as operações que envolvam uma complexidade maior.

O SQA é um modelo de replicação tolerante a faltas bizantinas que assegura que o sistema permanece correcto na presença de $n$ réplicas, sendo $n = 3f + 1$, e $f$ o número máximo de máquinas que podem falhar de forma bizantina. Se este pressuposto for satisfeito o SQA garante duas propriedades bastante importantes em sistemas bizantinos:

**Linearizability:** O sistema executa as operações de modo a que o sistema aparenta ser acedido sequencialmente;

**Wait-freedom:** Os pedidos realizados por clientes correctos são efectuados no sistema independentemente do comportamento de outros clientes.

A primeira é uma propriedade de *safety* que garante que o conjunto de réplicas comporta-se como um sistema centralizado executando uma mensagem de cada vez. A segunda propriedade é uma propriedade *liveness* importante para garantir a correcta terminação de todas as operações, ou seja, mesmo em cenários em que os servidores se encontrem sobrecarregados (ex: ataque *Denial of Service*), é assegurado que um cliente irá, eventualmente, receber resposta ao seu pedido.

O SQA permite replicar objectos sendo que, sobre estes, é possível realizar três tipos de operações distintas:

**Escrita:** O estado do objecto é alterado para o valor recebido como parâmetro;

**Leitura:** O estado do objecto é retornado;

**Actualização (*Read-Write-Modify*):** O estado do objecto é modificado de acordo com os parâmetros recebidos e o estado actual do objecto.

As duas primeiras operações são implementadas através de sistemas de quoruns bizantinos e, como consequência, estas operações podem ser construídas de forma assíncrona, não dependendo assim de condições optimistas, nem assumpções sobre tempos para terminar. A actualização recorre à máquina de estados replicada sendo que, por este motivo, requer sincronismo pois necessita de resolver consenso. Relativamente à complexidade de mensagens as operações de escrita e leitura apresentam complexidade de $O(n)$, enquanto que a actualização necessita de $O(n^2)$.

Os protocolos de escrita e leitura são adaptações dos protocolos apresentados em [16] enquanto que a actualização é baseada no algoritmo de ordenação de mensagens do CL-BFT [5]. O SQA apresenta-se como o primeiro sistema a juntar sistemas de quoruns bizantinos com máquina de estados replicada para realizar operações de diferentes semânticas. Consoante a operação em questão o sistema comporta-se como um sistema de quoruns bizantino ou como um máquina de estados replicada. No entanto, conhecendo a forma como as duas abordagens funcionam, torna-se evidente que é necessário contornar certos problemas que são levantados ao se combinar estas duas técnicas. Tipicamente, as operações de escrita e leitura não foram concebidas para operarem concorrentemente com operações de actualização, sendo que torna-se essencial possibilitar realizar escritas ou leituras mesmo quando os vários servidores se encontram em estados diferentes (provocados por uma actualização simultânea). Por outro lado, também é necessário assegurar que a actualização é executada mesmo quando os servidores apresentam estados diferentes. Este problema não é trivial visto que o sistema está sujeito a faltas bizantinas. Por exemplo, um cliente malicioso pode tentar realizar uma operação apenas em alguns servidores tentando causar uma falha no sistema por inconsistência. Torna-se então necessário encontrar mecanismos que detectem o estado inconsistente das réplicas sendo possível capturar o estado mais recente e, de alguma maneira, assegurar que todos os servidores correctos cheguem a um estado consistente. Outro problema que está inerente à junção de sistemas de máquina de estados com sistemas de quoruns é a forma como o sistema deve executar as operações de forma a que pareçam ser realizadas sequencialmente (*linearizability*).

Anteriormente ao SQA, foi desenvolvido o *LBTS* [3], que concretiza um espaço de tuplos tolerante a faltas bizantinas. O *LBTS* foi o primeiro modelo a juntar sistemas de quoruns bizantinos com máquina de estados replicada para realizar diferentes operações complexidades, no entanto, e ao contrário do SQA, o *LBTS* não pode ser utilizado para concretizar um serviço generalista pois foi desenvolvido especificamente para actuar num espaço de tuplos.

Nas próximas secções irá ser explicado em detalhe a forma como as operações do SQA funcionam assim como os métodos usados para resolver os problemas discutidos anteriormente.

## 3.3 Protocolos

Esta secção descreve os três protocolos concretizados no SQA. Todos eles são iniciados pelo cliente que envia um pedido para todos os servidores. Um aspecto importante a definir é o estado dos servidores. Cada servidor tem o seu próprio estado que é formado pelo conjunto de estados dos diferentes objectos. O estado de um objecto $o$ é formado por um tuplo $\langle S_o, t_o, C_o \rangle$, sendo $S_o$, o valor actual do objecto; $t_o$, a estampilha temporal que contem o número de versão actual do objecto $o$; e $C_o$, o certificado que valida o estado do objecto. Antes de iniciar a descrição dos protocolos é essencial definir o conceito de certificado pois é bastante usado ao longo dos protocolos. A próxima secção permite ficar com uma noção deste conceito.

De notar que o SQA está concebido para operar sobre múltiplos objectos replicados, no entanto, e para simplificar a descrição do protocolos, nesta secção apenas é considerado um objecto replicado.

### 3.3.1 Certificados

Para a execução de certos passos dos protocolos, o SQA utiliza o conceito de certificado de modo a ser possível verificar a validade da informação relativa dos objectos [17]. Neste contexto, um certificado é um conjunto de $n - f$ mensagens correctamente assinadas provando que para um par valor, $v$, e respectiva estampilha temporal, $t$, pelo menos $f + 1$ servidores correctos aceitaram a alteração do estado antigo para o novo estado.

O certificado permite que cada processo consiga verificar as acções realizadas por outros processos, sendo que, para que isto se torne possível, é necessário que cada objecto manipulável tenha o seu próprio certificado.

No SQA este conceito de certificado é definido por *update certificate* e é formado durante a execução do protocolo de escrita ou do protocolo de actualização. Relativamente à escrita, o certificado poderá ser formado por mensagens do tipo PREPARED ou TS. Em relação à actualização o certificado é formado pelas mensagens COMMIT. Enquanto que os certificados formados durante o protocolo de escrita dão legitimidade ao cliente para efectuar escritas, os formados durante a actualização permitem verificar a validade das mudanças de estado de cada objecto.

Nas próximas secções são explicados os protocolos e também o modo como os certificados são formados em mais detalhe.

### 3.3.2 Escrita

Como dito anteriormente, a operação de escrita permite que um cliente altere o valor do estado actual de um objecto. Esta operação pode ser efectuada em dois ou três acessos ao sistema consoante a consistência do estado do objecto em questão pelos vários servidores.

Conforme ilustrado na Figura 3.1, para executar o protocolo de escrita é necessário seguir os seguintes passos:

1. O cliente inicia o protocolo enviando uma mensagem do tipo $\langle \text{GET-TS}, \mathsf{H}(v) \rangle$ para todos os servidores, sendo $v$ o valor que se pretende escrever. Estes, ao receberem uma mensagem deste tipo, respondem com a mensagem $\langle \langle \text{TS}, \mathsf{H}(v), t_s \rangle_{\sigma_s}, C_s \rangle$. O cliente espera por $n - f$ respostas TS sendo que, cada mensagem só é validada se a assinatura estiver correcta e se a máxima estampilha temporal encontrada no certificado $C_s$ for igual a $t_s - 1$.

2. Após receber $n-f$ mensagens válidas do tipo TS, o cliente verifica se os servidores responderam com estampilhas iguais. É nesta verificação que é decidido se a escrita é executada em dois ou em três acessos. Se o servidores responderem com a mesma estampilha então a escrita é optimizada (dois acessos). Caso contrário é operação é processada como uma escrita normal (três acessos).

   **Escrita Optimizada** As primeiras $n - f$ respostas TS de servidores diferentes recebidas contêm a mesma a mesma estampilha, $t_v$, logo formam um certificado, $C_v$, necessário para executar o passo 3;

   **Escrita Normal** Como as estampilhas temporais recebidas são diferentes é necessário o cliente capturar a estampilha mais elevada. Para isso, o cliente envia a mensagem $\langle \text{PREPARE}, \mathsf{H}(v), t_s + 1, C_s \rangle$ para todos os servidores, sendo $t_s$ o valor mais elevado encontrado. Estes, ao receberem esta mensagem, verificam se a estampilha recebida é uma unidade superior à validada no $C_s$ e, enviam a mensagem $\langle \text{PREPARED}, t_s + 1, \mathsf{H}(v) \rangle_\sigma$ para o cliente. Este espera por $n - f$ respostas do tipo PREPARED, verificando a assinatura, formando assim um *update certificate*, $C_v$, necessário para o último passo.

3. O cliente prepara uma nova estampilha, $t_v$. No caso de ser uma escrita optimizada, $t_v$ é definido por um mais a estampilha temporal recebida nas mensagens no passo 1 . No caso de ser uma escrita normal, $t_v$ é igual à estampilha preparada no passo 2. Após este processo o cliente envia uma mensagem $\langle \text{WRITE}, t_v, v, C_v \rangle$ para todos os servidores. De notar que este $C_v$ é o *updade certificate* formado no passo anterior. Os servidores apenas aceitam mensagens WRITE se $t_v$ for maior que a sua estampilha actual e se $C_v$ for um *update certificate* válido para o par ($v$,$t_v$). Se a mensagem for validada, cada servidor efectua a operação no objecto em questão. Esta operação actualiza o estado do objecto para ($v$,$t_v$) assim como o seu certificado. Por fim, é enviada uma mensagem do tipo $\langle \text{ACK}, t_v \rangle_{\sigma_s}$ para o cliente. Ao receber $n - f$ respostas, o cliente termina a operação com a certeza de que o seu pedido foi processado pelo sistema.

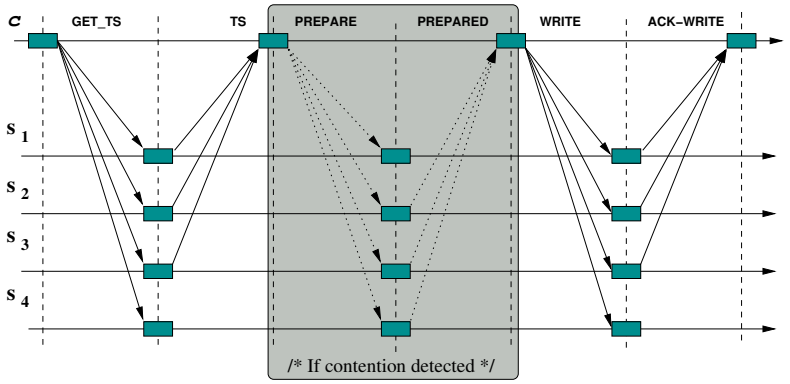


Figura 3.1: Protocolo de escrita.

Num cenário em que não há concorrência e todos os servidores são correctos, o cliente executa sempre o protocolo em dois acessos ao sistema. No entanto, isto já não se verifica no caso de haver mais de um cliente a efectuar uma escrita ou a efectuar uma actualização. Pode suceder que dois clientes correctos recebam estampilhas temporais diferentes devido a que certos servidores já se encontrem com um estado mais recente que outros. Esta situação não altera o correcto funcionamento do protocolo visto que o sistema está preparado para detectar estas possíveis inconsistências sendo efectuada uma escrita normal.

Os dois primeiros acessos são importantes para que seja possível que o cliente obtenha a estampilha temporal, $t_s$, do objecto que pretende manipular e construir um certificado válido para o par (valor, $t_s$).

Os certificados assumem uma grande importância para o funcionamento correcto deste protocolo. É através deles que os servidores conseguem proteger-se de clientes bizantinos pois, lembrando que cada certificado contem $n - f$ mensagens assinadas, os clientes não têm a capacidade de corromper ou criar novos certificados. Como os servidores só processam as operações de escrita se o *update certificate* recebido for válido torna-se impossível para um cliente bizantino realizar um escrita ”maliciosa”no sistema.

### Escritas consecutivas de um mesmo cliente

Como descrito anteriormente, a operação de escrita pode ser executada em dois ou três acessos ao sistema consoante a consistência dos estados dos objectos. O primeiro acesso permite que o cliente obtenha a estampilha temporal do objecto que pretende escrever, construir um certificado que valide a operação e, deste modo realizar a escrita com a última estampilha. Analisando esta operação constata-se que a primeira fase só é necessária em ambientes onde diversos clientes escrevem num mesmo objecto. Em objectos que tenham apenas um escritor, o cliente consegue definir a estampilha da próxima escrita como sendo a estampilha da sua última escrita realizada $t$ mais uma unidade. Assim, no caso do cliente já ter a estampilha $t$, de um certo objecto, ele tenta efectuar a escrita sem

realizar a primeira fase do protocolo enviando a mensagem WRITE já com o $t + 1$. Um servidor, ao receber esta mensagem, primeiro verifica se a sua estampilha actual para o objecto é $t$. Se sim, aceita a escrita. Caso contrário, $t + 1$ é uma estampilha antiga e portanto o servidor responde com uma mensagem informando o cliente da sua estampilha actual do objecto. Caso não seja detectada concorrência, a operação de escrita termina em apenas dois acessos ao sistema.

Para que esta optimização seja possível é necessário que o cliente consiga formar um certificado sem ter de executar a primeira fase da escrita, sendo que, o certificado é formado com as mensagens $ACK$ provenientes da última escrita. Esta optimização contribui bastante para o aumento de desempenho da operação de escrita, conforme será demonstrado no capítulo 5.

### 3.3.3 Leitura

A operação de leitura permite que um cliente leia o estado actual de um objecto. Esta operação tem a propriedade de retornar o valor mais recente independentemente da possível inconsistência dos estados dos servidores. Adicionalmente é necessário garantir que todas as leituras posteriores irão retornar o mesmo valor ou um mais recente (nunca um mais antigo). O protocolo está desenvolvido e preparado para garantir as propriedades referidas sendo que é necessário executar os seguintes passos:

1. O cliente inicia o processo enviando a mensagem $\langle$READ$\rangle$ para todos os servidores que enviam como resposta a mensagem $\langle$READ-REPLY$, t_s, v_s, C_s\rangle$ , sendo $t_s$ e $v_s$ o estado actual do objecto. O cliente espera até receber $n-f$ mensagens de servidores diferentes sendo que só são válidas se os $v_s$ recebidos tiverem o mesmo valor e se o certificado $C_s$ validar ($v_s$,$t_s$).

2. Após recepção de $n - f$ mensagens READ-REPLY válidas duas situações podem acontecer consoante a consistência das respostas:

   **Leitura Optimizada** Caso todas as mensagens tenham retornado o mesmo par ($v$,$t$), o valor $v$ é retornado como resultado da operação.

   **Leitura Optimizada** Caso os valores sejam diferentes, o cliente selecciona o par com a estampilha temporal mais elevada, ($v_s$,$t_s$), devidamente validada pelo seu $C_s$, e escreve o tuplo ($v_s$,$t_s$, $C_s$) para os servidores que apresentem um estado desactualizado. Este procedimento denomina-se por *writeback*. Quando um servidor recebe um *writeback*, altera o seu estado de acordo com o estado recebido enviando como resposta uma mensagem ACK_WB, retornando então $v_s$ como o resultado da leitura.

A Figura 3.2 ilustra os diversos passos de comunicação descritos.

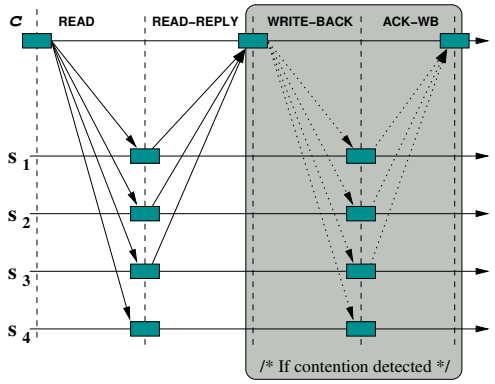


Figura 3.2: Protocolo de leitura.

Esta operação usa o conceito de *writeback* para garantir que futuras leituras retornam o estado mais actual. Internamente, o *writeback* é equivalente a efectuar o terceiro passo da escrita (PRE-PREPARE), com a diferença de que apenas os servidores desactualizados recebem essas mensagens. Num cenário onde não há concorrência, a leitura torna-se numa operação bastante simples sendo que os servidores apenas necessitam de retornar o estado do objecto. No entanto, num cenário real em que a leitura pode estar a ser processada simultaneamente com uma escrita ou como uma actualização, o *writeback* assegura que todos os servidores vão terminar esta operação com o mesmo estado.

### 3.3.4 Actualização

Este protocolo é o mais complexo das três operações suportadas pelo SQA. A operação de actualização (*Read-Modify-Write*) modifica o estado de um objecto com base no parâmetro recebido e no estado actual do objecto. Esta operação, que tem uma semântica mais forte que as escritas e leituras, pois, como cada operação tem de ter em consideração o estado anterior, é necessário que os pedidos sejam ordenados. Para isso, é necessário recorrer à máquina de estados replicada para a concretizar. Neste modelo, os pedidos são todos ordenados por ordem total sendo necessário envolver resolução de consenso. Por este facto, a actualização tem de ser processada num ambiente síncrono de modo a assegurar a terminação do consenso. Deste modo, como referido anteriormente, o SQA requer sincronia parcial relativamente a assumpções temporais.

A actualização foi construída com base numa versão modificada do protocolo CL-BFT [5], de modo que, o SQA tem a capacidade de ordenar os pedidos de forma que todos os servidores os executem pela mesma ordem. Adicionalmente, o sistema tem a capacidade de eleger um líder (que é responsável por propor números de sequência para as mensagens) e forma de detectar possíveis falhas no seu comportamento. Se necessário um novo líder pode ser eleito, recorrendo a um algoritmo de eleição, sendo que o estado global do sistema tem de ficar consistente no fim deste processo.

Para executar a o protocolo de actualização é necessário seguir os seguintes passos:

1. O cliente envia a mensagem $\langle$RMW, op$\rangle$ para todos os servidores. Esta mensagem indica a intenção do cliente realizar a operação *op* no sistema. Os servidores, ao receberem este pedido enviam a mensagem $\langle \text{STATE}, \langle \mathsf{H}(op), t_s, v_s \rangle_{\sigma_s}, C_s \rangle$ para o líder actual sendo $(t_s, v_s)$ o estado actual do objecto afectado pela operação *op*. O líder valida a mensagem verificando se a assinatura é correcta e verificando se o certificado $C_s$ corresponde ao par $(t_s, v_s)$. Na recepção de $n - f$ mensagens STATE válidas, o líder escolhe o par $(t_s, v_s)$ com a estampilha temporal mais elevada sendo este estado denominado por *curr_state*.

2. O líder executa a operação *op* com base no *curr_state* dando origem a um tuplo com os valores $\langle curr\_state, C_l, new\_state, r \rangle$. O *new_state* é composto pelo par $(t, v)$ sendo $t = t_s + 1$ e $v$ o novo valor do objecto resultante da operação. $C_l$ é o certificado que valida o novo estado e $r$ é a resposta a ser enviada para o cliente mais tarde. O tuplo descrito corresponde a parte da proposta efectuada pelo líder sendo que, de seguida, inicia o CL-BFT com algumas modificações.

3. Cada servidor, ao receber a mensagem PRE-PREPARE (relativa do CL-BFT) contendo a proposta formada no passo anterior e um número de sequência proposto, tem de efectuar as seguintes verificações para continuar a efectuar a actualização:

   - o *curr_state* é validado pelo $C_l$, ou seja, se o estado em que o líder aplicou a actualização é o último registado pelo sistema e, como consequência, está referido no certificado;
   - $t = t_g + 1$, sendo $t$ a estampilha temporal presente no *new_state* e $t_g$ a presente no *curr_state*;
   - $r$ e *new_state* são válidos, ou seja, é verificado se o novo estado e a resposta são um resultado válido tendo em conta o ultimo estado e a operação *op*;
   - as condições impostas pelo CL-BFT são cumpridas [5].

   Caso uma destas verificações seja falsa, a mensagem PRE-PREPARE não é válida sendo que o servidor em questão irá suspeitar do líder (situação discutida na próxima secção). De notar que, caso o servidor receba $n - f$ mensagens COMMIT de servidores diferentes continua a executar a actualização na mesma, significando que pelo menos $n - f$ servidores correctos aceitaram os valores propostos pelo líder.

4. Após receber $n - f$ mensagens COMMIT, resultantes da execução do CL-BFT, cada servidor define o par ($t_s$, $v_s$) de acordo com o novo estado presente na proposta e forma um novo certificado com base nas mensagens recebidas. Por fim, envia a mensagem $\langle RMW, r \rangle$ para o cliente que iniciou o protocolo, sendo $r$ a resposta formada pelo líder.

5. O cliente espera por $n - f$ respostas de servidores diferentes de modo a finalizar o protocolo e a retornar o resultado.

A Figura 3.3 ilustra os passos do protocolo descrito.

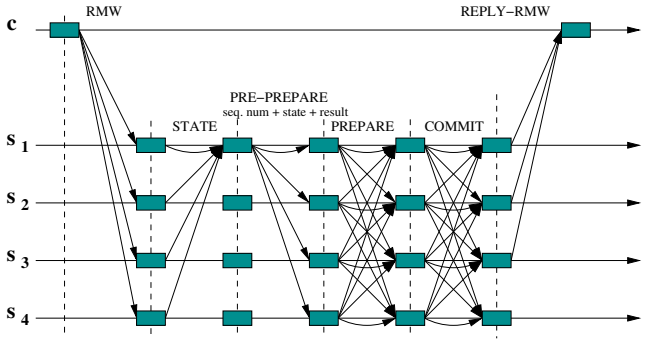


Figura 3.3: Protocolo de actualização.

Em ambientes em que há concorrência pode ser comum que servidores correctos apresentem estados diferentes sendo que é necessário assegurar que os pedidos são efectuados baseadas no estado mais recente. As mensagens STATE ganham uma importância bastante elevada pois é através delas que o líder consegue obter o estado mais recente. Adicionalmente este tipo de mensagem também é necessário durante a eleição de um novo líder. Este último cenário está abordado na próxima secção.

Uma característica bastante interessante, que não é verificado em outros sistemas BFT que usem máquina de estados replicada é a capacidade do sistema executar operações não deterministas. Este facto é possível pois apenas o líder necessita de executar os pedidos dos clientes, sendo que os outros servidores actualizam o seu estado para o estado recebido na proposta enviada pelo líder.

### Líder Faltoso

Como descrito anteriormente, para a execução deste protocolo é necessário recorrer a algoritmos de eleição de líder, sendo este responsável pela ordenação das mensagens.

Tendo em conta o modelo de faltas bizantinas assumido no SQA é de esperar que o líder do grupo de servidores seja um alvo bastante apetecível aos olhos de um *hacker*. No caso de um ataque bem sucedido, poderia ser fatal para o funcionamento correcto do sistema se este não estivesse preparado para lidar com um líder faltoso. Casos como o líder não propor ordem para uma mensagem ou propor ordens diferentes para a mesma mensagem são alguns cenários possíveis. Torna-se necessário ter mecanismos que permitam aos outros servidores ter a capacidade de detectar comportamentos estranhos por parte do líder, podendo então dar inicio a um procedimento de troca de líder.

Como o SQA recorre à biblioteca *Java Byzantine Paxos* (JBP)[1] para realizar difusão

[1] Versão modificada do CL-BFT explicada no próximo capítulo.

atómica em sistemas bizantinos, alguns destes pontos descritos acima já estão concretizados. Tal como no CL-BFT, o JBP requer sincronismo para assegurar a terminação da consenso, ou seja, há assumpções sobre o tempo máximo que cada passo do protocolo necessita. No JBP, os servidores baseiam-se nessas assumpções para associar um *timeout* a cada mensagem por ordenar. Caso o *timeout* para uma mensagem expire sem que esta tenha sido ordenada, os servidores irão suspeitar do líder dando origem a um procedimento com intenção de eleger um novo líder. Note-se que suspeitar é diferente de detectar. Pensando num cenário em que a rede se encontre congestionada, um líder correcto pode ser suspeitado pelos outros processos apenas porque algumas mensagens estão a demorar mais tempo do que o esperado. O mesmo procedimento de eleição de novo líder é invocado no caso do resolução consenso não obter progresso. Este cenário poderá acontecer no caso de um líder faltoso propor ordens diferentes para as mensagens.

Estas características juntamente com a verificação dos parâmetros da mensagem PRE-PREPARE (descrito na secção anterior) permitem que o SQA obtenha uma certa resiliência a líderes faltosos sendo que apenas é necessário assegurar a consistência dos estados dos servidores após a mudança de líder. Para isso, quando um novo líder é eleito, todos os servidores enviam para ele a mensagem $\langle \text{STATE}, \langle \mathsf{H}(op), t_s, v_s \rangle_{\sigma_s}, C_s \rangle$ sendo $op$ a operação que tinha ficado pendente. Na prática, esta mensagem é enviada juntamente com uma mensagem do JBP indicando a mudança de vista (equivalente à mudança de vista do CL-BFT). Deste modo, o novo líder tem a capacidade de escolher o estado válido mais recente podendo de seguida continuar a executar pedidos, incluindo os pedidos que tinham ficado pendentes.

### Actualização Optimizada

O primeiro passo do protocolo de actualização é o envio dos estados para o servidor líder. Desta forma o líder tem a certeza do estado global do sistema antes de efectuar a operação. Esta abordagem é pessimista pois admite que os servidores encontram-se sempre em estados diferentes. Isto é possível em cenários em que há permanentemente operações de escrita e actualizações em simultâneo, no entanto, acaba por penalizar execuções na qual os servidores estão em concordância.

Uma outra abordagem pode ser seguida: supor que os servidores encontram-se sempre com o mesmo estado e, só na eventualidade de serem detectadas discrepâncias entre os estados dos diferentes servidores, é que o líder recolhe a informação necessária para escolher o estado mais recente.

Sendo assim, a actualização pode ser optimizada adaptando o protocolo para seguir a segunda abordagem. Conforme ilustrado na Figura 3.4, em vez dos servidores enviarem a mensagem STATE no inicio do protocolo, o líder, ao receber um pedido de um cliente, calcula o valor resultante da operação $op$ com base no seu estado enviando a mensagem PRE-PREPARE para o grupo de servidores. As outras réplicas apenas aceitam a proposta

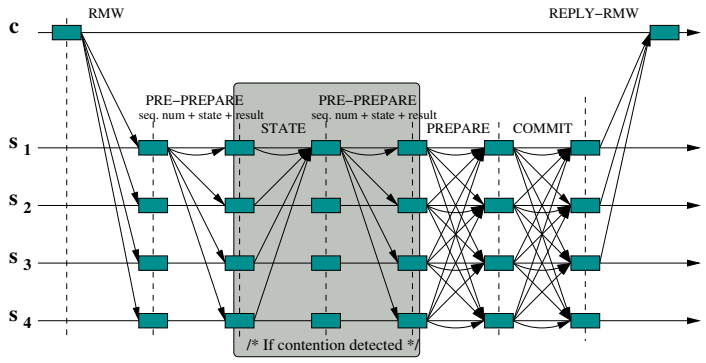


Figura 3.4: Protocolo de actualização optimizado.

caso a estampilha temporal recebida seja maior ou igual à sua estampilha actual. Se todos os servidores aceitarem a proposta, o protocolo continua a decorrer como suposto. Caso haja algum servidor que não considere a proposta válida, envia a mensagem STATE para o líder que irá esperar por $n - f$ mensagens STATE continuando posteriormente a execução normal do protocolo.

Esta optimização é óptima se considerarmos que não há contenção, ou seja, se não houver operações concorrentes (e como consequência o estado dos servidores são iguais) pois retira um acesso ao sistema no protocolo de actualização descrito na secção anterior.

## 3.4 Avaliação dos protocolos

Na secção anterior foram apresentados os três protocolos que o SQA propõe: escrita, leitura e actualização [4]. Nesta secção é apresentado um estudo sobre o desempenho destes protocolos. A Tabela 3.1 apresenta o número de passos de comunicação (estimativa de latência) e a complexidade de mensagens (que influencia a escalabilidade) destes protocolos.

| Protocolo | Número de passos | | Complexidade de Mensagem | Sincronismo |
|---|---|---|---|---|
| | Melhor Caso | Pior Caso | | |
| Leitura | 2 | 4 | $O(n)$ | Não |
| Escrita | 2 | 6 | $O(n)$ | Não |
| Actualização | 4 | 7 | $O(n^2)$ | Sim |

Tabela 3.1: Resumo dos protocolos de escrita, leitura e actualização sobre o número de passos necessários no melhor e pior caso, complexidade de mensagens e necessidade de assumpções temporais.

A leitura, tipicamente, é executada apenas com dois passos de comunicação (o mínimo que se consegue obter pois é necessário um pedido e uma resposta), sendo que, se for realizado um *writeback* é preciso mais dois passos. Já a escrita, no melhor caso, consegue terminar com o mesmo número de passos que uma simples leitura. Isto é possível devido

à optimização efectuada no protocolo de escrita, que permite que um cliente realize várias escritas consecutivas sem executar o pedido de obtenção da estampilha temporal. Se for detectada concorrência, a escrita pode chegar até aos seis passos de comunicação. A actualização, no melhor caso, consegue ser executada em quatro passos, que é o mínimo de passos possível para resolver o consenso em ambientes parcialmente síncronos [18]. De notar que é possível realizar actualizações em quatro passos recorrendo à optimização de *Tentative Execution*[2] descrita na secção 6 de [5]. No pior caso, desconsiderando eventuais trocas de líder, é necessário realizar sete passos de comunicação.

Em termos de complexidade de mensagens dos protocolos, é de notar o aumento considerável da actualização em relação à escrita e leitura. Isto é explicado pois a actualização utiliza a abordagem da máquina de estados replicada, ao invés de sistemas de quoruns bizantinos, de tal forma, que tem de recorrer ao algoritmo de consenso aumentando assim a complexidade de $O(n)$ para $O(n^2)$.

Pelo mesmo motivo, a actualização requer assumpções temporais para assegurar a terminação do consenso, ao contrário das escritas e leituras que têm a capacidade para actuar em ambientes assíncronos. Estas características dos protocolos tornam o SQA num modelo parcialmente síncrono.

## 3.5 Considerações Finais

Neste capitulo, foi apresentado e discutido o Sistemas de Quoruns Activos, um novo sistema de replicação da família dos sistemas tolerantes a faltas bizantinas, que foi o primeiro modelo a utilizar sistemas de quoruns bizantinos e máquina de estados replicada para efectuar operações com diferentes semânticas e consistências. Ao contrário da maioria dos sistemas existentes, que apenas admitem duas operações (escritas e leituras), o SQA fornece protocolos de leitura, escrita e actualização sendo que, a separação das ultimas duas operações é bastante interessante para aplicações que concretizem operações que considerem o estado antigo do objecto em questão, e outras operações que não necessitem desse factor.

Com base neste modelo de replicação, foi desenvolvida a biblioteca Objectos de Serviço Confiável que concretiza os protocolos propostos pelo SQA, sendo esta apresentada no próximo capítulo.

[2]Esta optimização permite que os servidores respondam aos clientes com a mensagem PREPARE, sendo que estes esperam por $n - f$ mensagens desse tipo para retornarem o resultado.

# Capítulo 4

# Concretização da biblioteca Objectos de Serviço Confiáveis

Este capítulo descreve o desenvolvimento da biblioteca **Objectos de Serviço Confiáveis** (OSC), que concretiza a proposta Sistemas de Quoruns Activos descrito no capítulo anterior, assim como o modelo de programação e técnicas usadas para a sua concretização. Serão também discutidas optimizações que foram efectuadas aos protocolos de escrita, leitura e actualização originais.

## 4.1 Considerações gerais

A biblioteca OSC foi desenvolvida na linguagem de programação Java sendo que foi necessário recorrer frequentemente a funcionalidades da API de concorrência e criptografia disponíveis. A comunicação foi concretizada usando como recurso uma biblioteca denominada *Communication System*[1]. Esta biblioteca implementa *sockets* TCP, sendo as mensagens autenticadas através de MACs (*Message Authentication Codes*) que utilizam o algoritmo SHA1.

Relativamente à concretização dos protocolos, as operações de leitura e escrita foram implementadas de raiz enquanto que o protocolo de actualização foi desenvolvido sobre o *Java Byzantine Paxos*[1] (descrito na próxima secção). Para a realização das assinaturas de chave pública foi utilizado a biblioteca *Crypto++* que implementa o algoritmo *ESIGN*, muito mais eficiente que o RSA, usualmente empregado para este fim. Como o *Crypto++* é concretizado na linguagem C++, foi necessário recorrer à tecnologia *Java Native Interface* que permite integrar um programa em Java com código escrito noutra linguagem.

[1] O *Communication System* e o JBP podem ser obtidos livremente em `http://www.navigators.di.fc.ul.pt/software/jitt/jbp.html`.

## 4.2 Java Byzantine Paxos

O *Java Byzantine Paxos* (JBP) é uma protocolo de difusão atómica tolerante a faltas bizantinas baseado no protocolo CL-BFT [5]. Tal como na proposta de Castro e Liskov, o JBP implementa o algoritmo Paxos at War [28], com modificações para efectuar difusão atómica com ordem total sobre o algoritmo de consenso. No JBP também existe um líder responsável por propor uma ordem a cada mensagem sendo que, caso o comportamento do líder não esteja de acordo com o esperado, pode haver uma eleição de um novo líder.

O principal objectivo do JBP é de fornecer uma biblioteca completa e modular de um protocolo de acordo baseado no paxos bizantino. O resultado final é similar ao CL-BFT, porém concretizado na linguagem de programação Java, ao invés de C.

## 4.3 Arquitectura

A biblioteca OSC foi desenvolvida com o intuito de criar uma biblioteca de replicação tolerante a faltas bizantinas possibilitando a criação de sistemas generalistas. Para suportar uma ampla gama de objectos, para os mais diversos serviços, foi desenvolvido um modelo de objectos com uma interface simples, tanto a nível da realização de pedidos pelos clientes, como no lado do servidor e na gestão dos objectos.

Como referido anteriormente, a biblioteca OSC faz uso do *Communication System* para realizar a comunicação e do JBP (do lado do servidor) para realizar a ordenação de mensagens quando se trata de actualizações, tal como ilustrado na Figura 4.1.

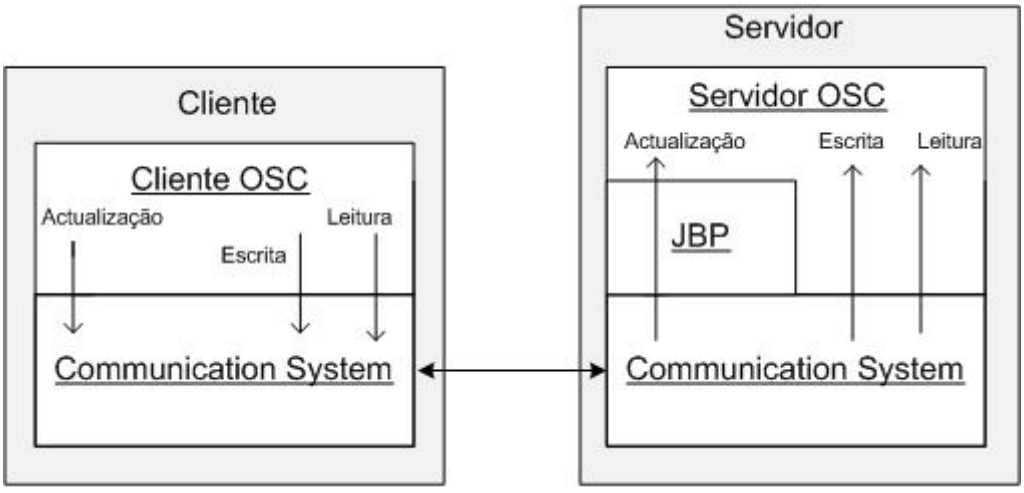


Figura 4.1: Arquitectura do cliente e do servidor da biblioteca OSC.

É importante notar que, apesar de na Figura 4.1 apenas estar representado um servidor, para que o sistema siga a especificação do SQA, é necessário um conjunto de servidores igual a $n = 3f + 1$.

### 4.3.1 Cliente

As aplicações clientes utilizam apenas uma classe básica (`AQSClient`) para aceder aos objectos suportados pelo servidor. Esta classe fornece métodos que permitem invocar os três protocolos descritos na secção anterior tal como descrito na Tabela 4.1.

| Operação | Método OSC |
|---|---|
| Leitura | *startRead(int idObject, Object optional);* |
| Escrita | *startWrite(Object o, int idObject, Object optional);* |
| Actualização | *startRMW(Object op, int idObject);* |

Tabela 4.1: Operações do SQA e métodos do OSC correspondentes.

Todos os métodos levam um identificador numérico do objecto a ser invocado e, nos métodos que iniciam as operações de escrita e leitura foi incluido um parâmetro opcional pois, por vezes, certas aplicações têm de especificar parâmetros adicionais nos seus pedidos. No caso da actualização não é necessário pois o parâmetro *op* já permite especificar a operação integralmente. Todas as operações são efectuadas de modo assíncrono, sendo que a aplicação não bloqueia à espera da resposta do pedido.

Adicionalmente, existe uma interface Java chamada `AQSClientAppl` que contém os métodos descritos na Tabela 4.2.

| Operação | Final da invocação |
|---|---|
| Leitura | *readCompleted(Object o);* |
| Escrita | *writeCompleted();* |
| Actualização | *rmwCompleted(Object o);* |

Tabela 4.2: Métodos de terminação dos protocolos.

Esta interface fornece os métodos para que a aplicação cliente seja notificada pelo `AQSClient` acerca da conclusão dos seus pedidos. A ideia é que qualquer aplicação cliente que utilize a biblioteca OSC implemente esta interface e registe-se com um *handler* na classe `AQSClient`.

Apesar de, como referido, as invocações serem assíncronas, é possível que uma aplicação concretize um serviço com pedidos síncronos, sendo que, neste caso, será a própria aplicação que terá de gerir os *locks* associados ao processo.

### 4.3.2 Servidor

Do lado de servidor é importante definir o modelo através do qual os vários tipos de objectos da biblioteca OSC são suportados. Para concretizar a ideia de um sistema generalista que se adapte a qualquer tipo de serviço, foi necessário especificar um certo padrão que os objectos têm de seguir. O ponto fundamental do sistema é uma classe abstracta, `AQSObject`, que contem um identificador único e armazena o estado completo

do objecto, ou seja, um par valor, estampilha temporal e respectivo certificado. Esta classe define um conjunto de métodos abstractos essenciais para a futura iteração entre os servidores e os objectos. Todos os objectos de serviço confiáveis são extensões da classe `AQSObject` de modo a obterem a interface que permita a manipulação das suas variáveis.

Para gerir o ciclo de vida destes objectos foi criada uma classe *singleton* chamada `AQSObjectManager`. Esta classe prevê métodos para criar e destruir os diferentes objectos presentes no sistema. É através dela que os servidores efectuam as operações e manipulam os objectos. Todos os objectos têm acesso ao `AQSObjectManager` e portanto são capazes de criar novos objectos e/ou destruir objectos activos.

A Figura 4.2 representa o modelo de classes dos objectos da biblioteca OSC.

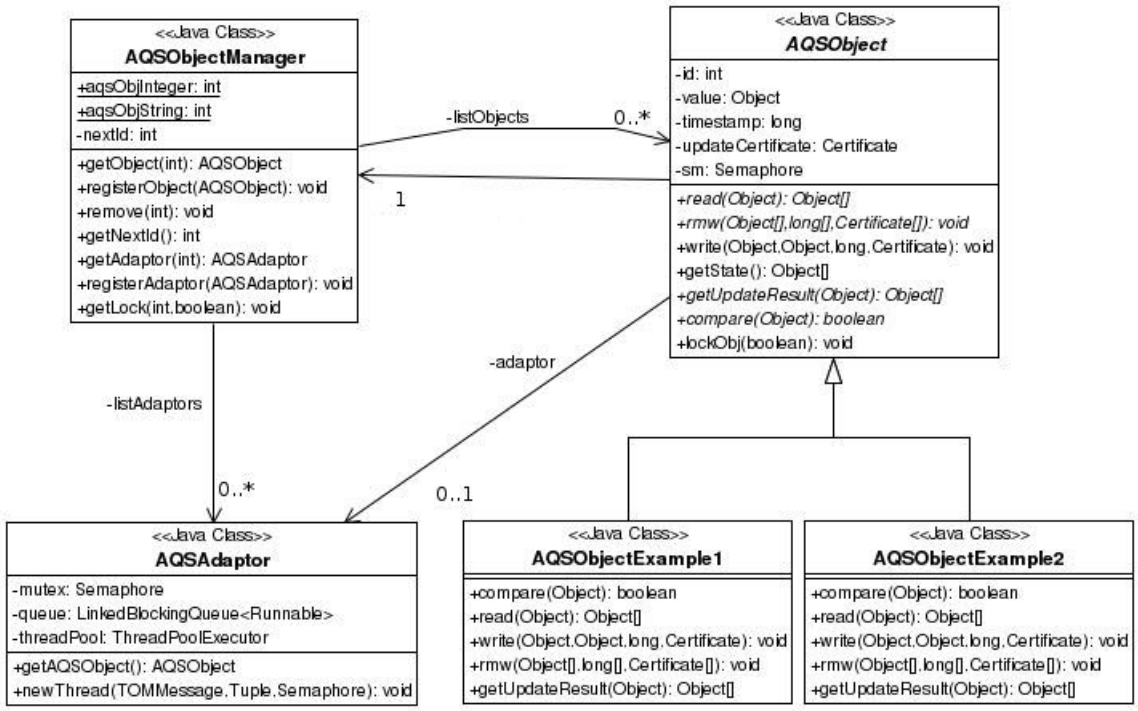


Figura 4.2: Modelo de objectos OSC com duas classes exemplos de objectos que estendem a classe `AQSObject`.

De notar que existe uma classe chamada `AQSAdaptor` que possibilita que os objectos OSC suportem o uso de *multithreading*. Este assunto será discutido numa secção posterior.

Um dos requisitos dos objectos do sistema é que têm de estar preparados para acessos concorrentes em um mesmo servidor. É possível que várias operações se realizem ao mesmo tempo e, para garantir a consistência dos dados, é necessário usar um modelo que controle o acesso concorrente aos objectos. Foi adoptado um sistema de *locks* de leitura/escrita sendo que, cada objecto tem o seu próprio *lock* e é o gestor de objectos que solicita o acesso ao objecto em questão. Caso seja detectado um acesso concorrente, o

pedido é bloqueado até o objecto estar disponível. De notar que estes *locks* são locais em cada réplica, ou seja, a sincronização apenas é efectuada entre as *threads* de um mesmo servidor, e não entre servidores.

Os acessos aos objectos, têm de ser condicionados pelos *locks*, quando se trata de acessos para realizar escritas ou actualizações concorrentes entre si. As leituras apenas têm de ser bloqueadas quando há concorrência com escritas ou actualizações. Entre leituras não é necessário aplicar o bloqueio visto que não há alteração do estado dos objectos. Esta abordagem de *locks* individuais permite que o acesso a cada objecto seja independente, garantindo assim uma maior disponibilidade dos mesmos.

## 4.4 Optimizações

Durante o processo de desenvolvimento da biblioteca OSC várias optimizações aos protocolos originais do SQA foram pensadas e concretizadas. Algumas delas resultaram num aumento de desempenho considerável. Nesta secção essas optimizações são discutidas.

### 4.4.1 Batching de mensagens

Recordando o protocolo de actualização, é necessário que o líder ordene as mensagem sendo que, este passo só é possível através da execução de algoritmos de consenso. Sempre que um pedido de actualização é solicitado, é iniciada uma nova instância do consenso o que faz com que cada pedido de um cliente tenha uma complexidade de mensagens na ordem de $O(n^2)$. Se imaginarmos um cenário em que existem vários clientes a efectuar pedidos de actualização, facilmente a rede ficaria congestionada devido a um grande número de mensagens trocadas entre os servidores. A optimização de realizar *batching* de mensagens permite que os vários pedidos dos clientes sejam agrupados de tal forma que o líder propõe a ordem para aquele grupo de mensagens em uma única execução do algoritmo de consenso. Desta forma, para cada agrupamento, apenas é necessário executar uma instância do consenso.

Esta optimização foi inicialmente proposta em [5] e já se encontrava concretizada no JBP. No entanto, o JBP só tem em consideração a ordem com que as mensagens são propostas. No SQA foi necessário estender essa optimização para a semântica da validade e consistência dos estados dos objectos (descrito no capítulo 3) para que o protocolo de actualização seja efectuado correctamente.

Se existirem várias operações, para um mesmo objecto, a serem ordenadas em um *batching*, é apenas enviada uma cópia do estado inicial (antes das actualizações serem executadas) e uma cópia do estado final do objecto (após a execução de toda sequência de operações) na mensagem PRE-PREPARE. Além disso, o sistema suporta *batching* de operações a serem executadas em objectos diferentes. Conforme será visto no capítulo 5, este tipo de optimização melhora em muito o *throughput* máximo do sistema pois,

o número de mensagens recebidas e enviadas e o número de instâncias de consenso necessárias baixam consideravelmente.

### 4.4.2 Sínteses de mensagens PAXOS (Weak, Strong, Decide)

No protocolo de actualização, é necessário que o líder envie para todas as réplicas a mensagem de PRE-PREPARE que pode ter dimensões elevadas devido a conter a proposta com o número de sequência juntamente com o estado do objecto em questão. O tamanho desta mensagem ainda pode ser maior considerando a optimização de *batching* discutida que agrupa vários pedidos. Após a formação desta mensagem é iniciado o algoritmo que irá desencadear o envio e recepção de um conjunto de mensagens por todas as réplicas, sendo que essas mensagens necessitam de conter a proposta. Visto que este processo pode ser penoso para o desempenho geral da actualização, é efectuada um redução na dimensão das mensagens do PAXOS bizantino, através do envio do *hash* da proposta, ao invés do envio da proposta integral (esta optimização também está presente em [5]). O impacto que esta optimização tem no tamanho das mensagens está apresentado na Tabela 4.3.

| Numero de pedidos | Propose | Weak, Strong, Decide |
|---|---|---|
| 1 | 515 | 87 |
| 2 | 660 | 111 |
| 3 | 805 | 135 |

Tabela 4.3: Tamanho das mensagens de *Propose*, em bytes, em comparação com as mensagens do PAXOS (Weak, Strong e Deceide) para objectos de 4 bytes, variando entre 1 e 3 pedidos para objectos diferentes.

### 4.4.3 Vectores de MACS em vez de assinaturas

No protocolo de escrita, é usada criptografia assimétrica em todas as mensagens que os servidores enviam para os clientes. A criptografia assimétrica garante propriedades muito fortes permitindo que os clientes consigam verificar a legitimidade das mensagens (não-repudiação). Por outro lado, o uso deste tipo de criptografia tem um impacto elevado no desempenho do protocolo.

Surge a necessidade de tentar reduzir ao máximo o uso de criptografia assimétrica tendo em consideração a não violação da especificação do sistema. Após uma análise ao protocolo de escrita, foi constatado que as mensagens ACK não necessitam de ter propriedades tão fortes como as asseguradas pela criptografia assimétrica. O cliente espera por $n - f$ mensagens ACK para terminar o protocolo sendo que, este tipo de mensagens é formado pela estampilha temporal actual de cada servidor. Como discutido na optimização anterior, estas mensagens ACK servem para a obtenção de um novo certificado. O cliente não necessita de verificar a assinatura destas mensagens neste processo

pois, numa escrita futura, os servidores irão verificar a validade desse certificado assim como das mensagens que o formam. Neste caso, o uso de assinaturas não é imperativo pois existem alternativas para que o grupo de servidores valide as suas próprias mensagens.

Com base nesta ultima valência, foi decidido utilizar o conceito de autenticador de mensagens (*Authenticator*) para validar as mensagens ACK. Esta optimização é bastante utilizada noutros protocolos de replicação que requerem mensagens assinadas [1, 5, 14], sendo que, um autenticador é um vector composto por $n - f$ MACs (*Message Authentication Code*) que provam a legitimidade de uma mensagem. Para que esta técnica seja exequível, é necessário que cada servidor possua uma chave secreta com cada um dos outros servidores. De notar que as mensagens PREPARED também utilizam vectores de MACs em vez de assinaturas, no entanto, relativamente às mensagens TS, já não é possível realizar esta optimização.

A Tabela 4.4 demonstra os tempos necessário para efectuar uma assinatura e vectores de MACs (com diferentes números de máquinas) de modo a se conseguir perceber, ao certo, o quanto é que esta optimização contribui para o aumento de desempenho.

| | Assinatura | Vector de MACs | | |
|---|---|---|---|---|
| | | n = 4 | n = 7 | n = 10 |
| Formar | 1.039 | 0.027 | 0.045 | 0.062 |
| Verificar | 0.536 | 0.011 | 0.011 | 0.011 |

Tabela 4.4: Tempo necessário em milissegundos para efectuar assinaturas e vectores de MACs para objectos de 172 bytes.

Como se pode observar, mesmo num cenário em que $n = 10$ consegue-se que os valores de processar e verificar vectores de MACS sejam bastante inferiores em relação ao tempo de efectuar assinaturas. Desta forma, num cenário sem concorrência, o protocolo de escrita é executado em apenas dois passos de comunicação, sem ser necessário recorrer à criptografia assimétrica, o que na pratica, é traduzido por um aumento de desempenho tanto a nível de latência com a nível de débito.

### 4.4.4 Actualizações sem enviar estado

Um dos aspectos mais penalizadores do protocolo de actualização do SQA, quando comparado a protocolos de ordenação como o CL-BFT, é o envio do estado dos objectos na formação da mensagem PRE-PREPARE pelo líder. Este, após realizar a operação pedida pelo cliente, forma uma proposta contendo o estado actual do objecto e o novo estado resultante da operação. Tendo em conta que os estados dos objectos podem ter dimensões elevadas e que o *batching* pode conter mensagens para diferentes objectos, percebe-se que esta carga adicional na mensagem acaba por ter um impacto não desprezível no desempenho do protocolo.

Para colmatar este aspecto foi concretizada uma optimização que permite que a actualização seja executada no sistema sem ser necessário o envio do estado na proposta enviada pelo líder. Ao invés disso, o líder envia a sua estampilha temporal actual, o *hash* do estado actual do objecto em questão, e um indicativo de como os outros servidores devem transitar para o novo estado. Este indicativo é totalmente controlado pela aplicação sobre o SQA. Para certas aplicações, essa variável pode ser de dimensões reduzidas, como por exemplo um *dif* entre o estado antigo e o novo estado, ou apenas um operação determinista de actualização a ser executada no estado antigo de modo a gerar o novo estado.

Com esta optimização, é dada liberdade à aplicação de definir como é que os diferentes servidores processam as operações e, em muitos casos, permite evitar o envio do estado dos objectos, verificando-se uma redução acentuada do tamanho das mensagem PRE-PREPARE.

## 4.5 Suporte a *multithreading*

Hoje em dia, os grandes distribuidores de *hardware* estão a adoptar uma abordagem diferente relativamente à construção de processadores (CPU's). Enquanto que antes o objectivo era de criar processadores cada vez mais potentes em termos de processamento, nos últimos anos têm surgido propostas de sucesso para novos modelos com dois processadores (*dual-core*) e, ainda mais recentemente, com quatro processadores (*quad-core*). Esta nova abordagem de agrupar diferentes núcleos de processamento permite repartir processamento por esses núcleos.

Foi com base nestes factos que se decidiu que a biblioteca OSC deveria ser concretizada num modelo *multithreading* do lado dos servidores, de modo a escalar o desempenho geral dos protocolos na presença de vários núcleos. Para isso, foi utilizado a técnica de *Thread Polling* que consiste na criação prévia de *threads* (é possível especificar o número de *threads* que se pretende criar) que, posteriormente, são alocadas para a execução de tarefas. As diferentes tarefas são organizadas numa fila, sendo que o *ThreadPoolExecutor* distribui essas tarefas para serem executadas pelas *threads* disponíveis.

A biblioteca OSC utiliza esta técnica de modo a conseguir que os diferentes servidores executem pedidos de vários clientes em simultâneo, fazendo uso de vários *cores*, melhorando significativamente o desempenho e a disponibilidade do sistema. As diferentes operações (escrita, leitura e actualização) foram alteradas de modo a estarem preparadas para lidar com este modelo de processamento. De seguida são apresentadas as adaptações necessárias nos diferentes protocolos.

**Leitura e Escrita.** Para a concretização do modelo *multi-thread* nestes dois protocolos, os passos realizados nos servidores são executados em *threads* diferentes. Relativamente à escrita, é importante constatar que a criação e verificação de assinaturas e vectores de MACs são executadas em *threads* separadas. Deste modo, num cenário

em que existam vários escritores concorrentes, as operações são realizadas em simultâneo. Caso as operações concorrentes afectem o mesmo objecto, o uso de *locks* de leitura/escrita garantem atomicidade no acesso ao estado do objecto, de forma a não existir nenhum conflito entre as operações.

A Figura 4.3 ilustra o modelo concretizado para os protocolos de escrita e leitura.

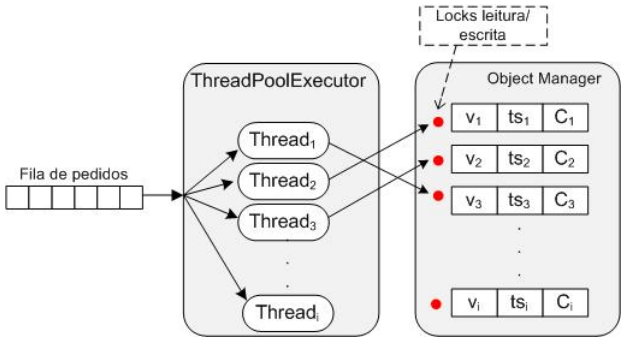


Figura 4.3: Modelo de *Thread Polling* usado nos protocolos de escrita e leitura.

**Actualização.** O passo mais exigente neste protocolo, em termos de processamento, é a formação da proposta pelo líder. Neste passo, o líder tem de obter os novos estados dos objectos resultantes das operações e enviar os indicativos de como é que os outros servidores transitam para o novo estado. Tendo em conta que é efectuado *batching* de mensagens, este processo afecta directamente o desempenho da actualização.

No sentido de melhorar este aspecto, foi aplicado a técnica de *Thread pooling* na formação das propostas, ou seja, o líder efectua cada pedido numa *thread* em separado e, quando os pedidos forem executados, a proposta é formada com os diferentes resultados obtidos. No entanto, há que ter em atenção as características da actualização que faz com que o resultado seja obtido com base no estado actual do objecto.

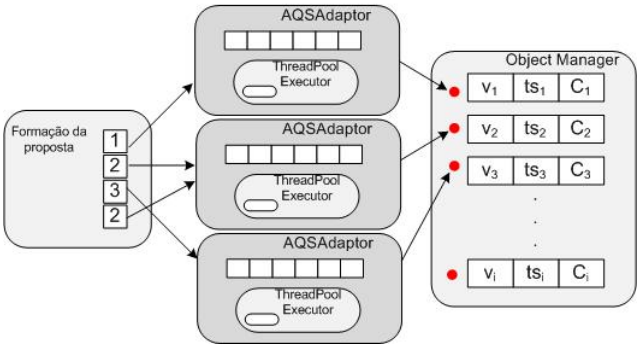


Figura 4.4: Modelo de *Thread Polling* usado no protocolo de actualização.

Como ilustrado anteriormente na Figura 4.2 (secção 4.3), foi desenvolvida a classe `AQSAdaptor` que serve como um adaptador do modelo *multi-thread* para um

`AQSObject` no protocolo de actualização. Cada `AQSObject` está associado a um `AQSAdaptor` que tem a tarefa de executar um *ThreadPoolExecutor* e obter o resultado da operação sobre o objecto em questão. Isto quer dizer que, cada objecto do sistema tem a sua própria *thread* de execução e a sua própria fila de operações de actualização por executar, tal como mostra a Figura 4.4.

Através destas adaptações nos protocolos de escrita, leitura e actualização é possível aproveitar e utilizar o vários *cores* dos sistemas computacionais actuais, de modo a aumentar a eficiência e disponibilidade do sistema.

## 4.6 Operações em Múltiplos Objectos

Uma funcionalidade bastante interessante que a biblioteca OSC oferece é a possibilidade de se realizar operações (do mesmo tipo) em múltiplos objectos. Este tipo de operações possibilitam que uma operação seja efectuada em um ou mais objectos apenas com um pedido. Esta característica possibilita que as operações de escrita e de actualização sejam efectuadas a um grupo de objectos de uma só vez. Por outro lado, a operação de leitura possibilita o retorno de um ou mais objectos no mesmo pedido.

As operações múltiplos objectos podem ser aproveitadas pelas aplicações com vista a reduzir o numero de pedidos necessários para efectuar operações especificas da aplicação.

## 4.7 Considerações Finais

A biblioteca Objectos de Serviço Confiáveis concretiza a proposta Sistemas de Quoruns Activos apresentada no capítulo anterior. O objectivo da OSC é fornecer uma biblioteca com uma interface simples de modo a facilitar a integração com qualquer tipo de serviço generalista. Esta biblioteca já concretiza certas optimizações, sendo que algumas delas foram inspiradas em optimizações concretizadas noutros sistemas BFT.

Uma das particularidades mais fortes desta biblioteca é o suporte da técnica de *multithreading* que permite distribuir as diversas operações em *threads* previamente criadas, de modo a aproveitar os vários núcleo de processamento.

Com esta versão da biblioteca OSC foram efectuados testes ao desempenho dos três protocolos desenvolvidos. Esses resultados são apresentados no próximo capítulo.

# Capítulo 5

# Avaliação Experimental da Biblioteca OSC

Este capítulo apresenta os resultados de desempenho dos protocolos SQA obtidos em LAN. Foram realizados testes relativamente à latência e ao número máximo de operações por segundo que o sistema consegue executar.

## 5.1 Ambiente

A não ser que seja dito o contrário, todos os experimentos foram efectuados num ambiente fechado, no qual todas as máquinas usadas são PCs Pentium-4 com processamento de 2.8 GHz com 2 GBs de RAM interligadas por um rede *gigabit*. Estas máquinas correm com a plataforma Sun JDK 1.6 sobre Linux 2.6.18.

Além disso, todos os experimentos consideram serviços nulos (operações de actualização que não requerem nenhum processamento de estado) com $n = 4$ servidores que toleram $f = 1$ faltas. Consideramos apenas este número mínimo de réplicas devido ao alto custo da manutenção da independência de falhas em sistemas tolerantes a intrusões [21].

## 5.2 Latência

Estes testes têm o objectivo de se obter a latência média para cada protocolo do SQA, sendo que, para os realizar, foi efectuada uma medição do tempo que demora um único cliente a efectuar 10.000 pedidos. Esses pedidos são enviados um de cada vez sempre que o anterior foi concluído. Foram realizados testes para objectos de diferentes dimensões com o intuito de analisar o impacto que essa mudança tem na execução dos protocolos. Os resultados são apresentados na Figura 5.1.

Os resultados mostram que os valores para os protocolos de escrita e leitura são de facto muito próximos, sendo que, para um objecto de 4 KBs, ambos os protocolos apre-

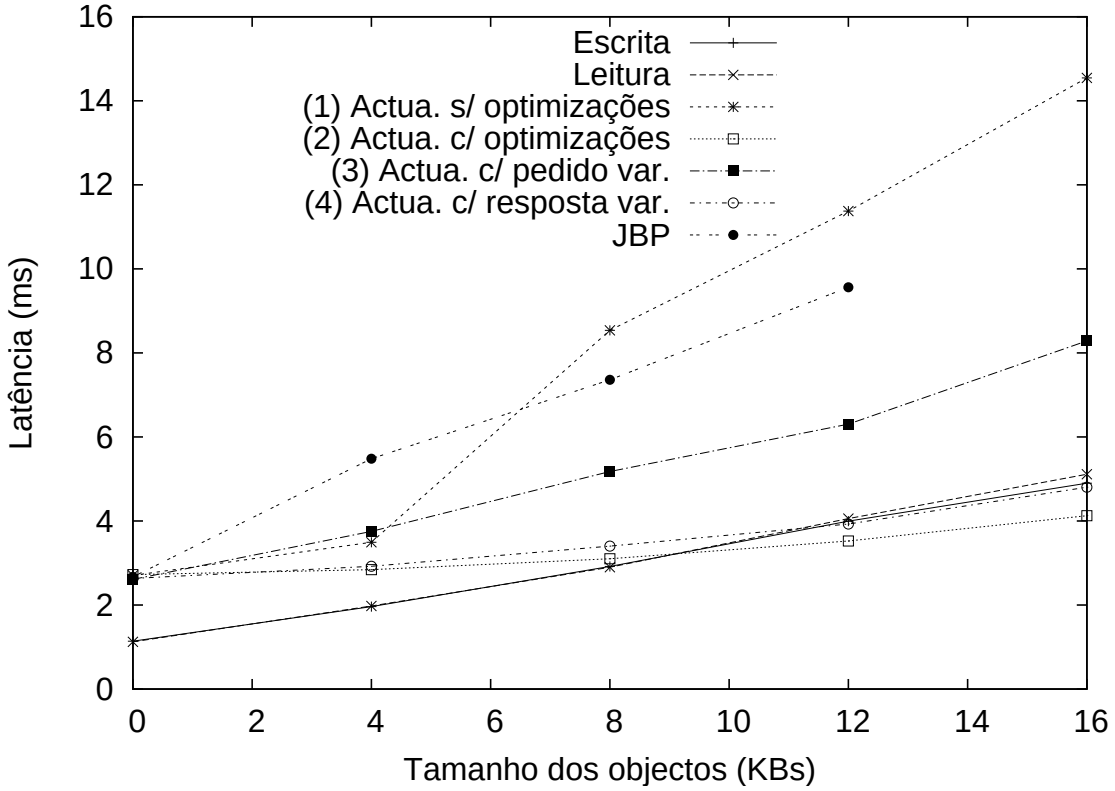


Figura 5.1: Latência dos vários protocolos pela biblioteca OSC para objectos de tamanho entre 4 bytes e 16 KBs.

sentam 1,9 milissegundos de latência. Este facto é explicado pela necessidade de construir um vector de MACs na escrita ($n-1$ MACs) e verificar vectores na leitura ($n-1$ MACs), sendo que, para isso cada processo demora aproximadamente 0,03 ms. Note que a escrita só consegue ser competitiva com a leitura devido às optimizações concretizadas (escritas consecutivas de um mesmo cliente e vector de MACs em vez de assinaturas) e ao facto de não serem consideradas escritas concorrentes neste experimento. Deste modo, o protocolo de escrita é sempre executado em apenas dois passos de comunicação sem ser necessário recorrer a criptografia assimétrica.

Relativamente ao protocolo de actualização, foram obtidos valores para uma primeira versão do protocolo e para a versão final que já concretiza a optimização do envio das proposta sem o estado dos objectos. Na primeira versão (1), ainda sem optimizações concretizadas, verifica-se que, com o aumento do tamanho dos objectos, há um grande impacto na latência obtida, em grande parte devido ao tamanho da proposta efectuada pelo líder. Entre objectos com dimensões de 4 KBs e de 16 KBs, a latência média varia entre 3,5 ms e 14,5 ms, respectivamente. Com a optimização da proposta sem estado (2), esse impacto negativo já não se verifica, deixando o protocolo de actualização com uma latência de 2,8 ms para objectos de 4 KBs, bastante baixa tendo em conta que está assente na máquina de estados replicada. Além dessas duas versões do protocolo de actualização,

são também apresentados resultados para outros padrões de operações de actualização: (3) uma em que um bloco de dados, do mesmo tamanho do objecto, é enviado no pedido do cliente; e outra versão (4) em que o estado do objecto é enviado pelos servidores na resposta. Conforme pode ser visto nos resultados, o sistema é muito mais sensível a pedidos grandes do que a respostas grandes, de tal forma que, para objectos de 4 KBs já se verifica uma diferença de quase 1 ms, aumentando este valor para 4 ms em objectos de 16 KBs. Estes resultados devem-se ao facto da proposta ser formada com o conteúdo do pedido, sendo interessante notar no impacto que o tamanho da proposta tem no desempenho do protocolo de actualização.

Os resultados da latência do JBP, foram obtidos em testes nos quais, tanto o pedido como a resposta são da dimensão dos objectos. Por este motivo, o JBP apresenta uma latência maior que as várias versões do protocolo de actualização. Apesar dos valores deste teste não serem conclusivos, dá para se perceber que a actualização obtém uma latência bastante baixa relativamente ao que máquina de estados replicada permite.

## 5.3 Throughput

O objectivo destes testes é estimar o número máximo de operações por segundo que o sistema consegue processar.

Para efectuar testes de carga, foram utilizadas 6 máquinas clientes cada uma executando, no máximo, 30 clientes lógicos enviando pedidos de 1 em 1 milissegundo, sem esperar pela resposta dos servidores. Deste modo, foi possível congestionar os servidores de forma a se obter valores máximos de débito. Estes testes foram realizados para objectos de 4 bytes e de 4 KBs.

As Figuras 5.2 e 5.3 apresentam os resultados obtidos para os diferentes protocolos fornecidos pelo OSC.

Ao contrário dos resultados para latência, os testes de *throughput* demonstram que o sistema tem uma capacidade muito maior de processar leituras do que escritas. Em objectos de 4 bytes, o sistema consegue processar até $33.557$ leituras por segundo e apenas $4.056$ escritas por segundo. Estes resultados são justificados pela necessidade dos servidores verificarem e construírem vectores de MACs em cada escrita, o que requer o cálculo de $2(n-1)$ *hashs* criptográficos. No caso da leitura, apesar desses vectores serem verificados, esta verificação é efectuada pelo cliente, o que não implica custos de processamento nos servidores. Este facto revela-se bastante penalizador para a escrita nos resultados obtidos.

Para a actualização, foram realizados experimentos com diferentes valores máximos para *batches* de mensagens e, conforme pode ser visto no gráfico da Figura 5.3, o número de actualizações por segundo suportadas pelo sistema aumenta consideravelmente conforme aumentamos o tamanho *batch*. A actualização atinge um máximo como *batch*-

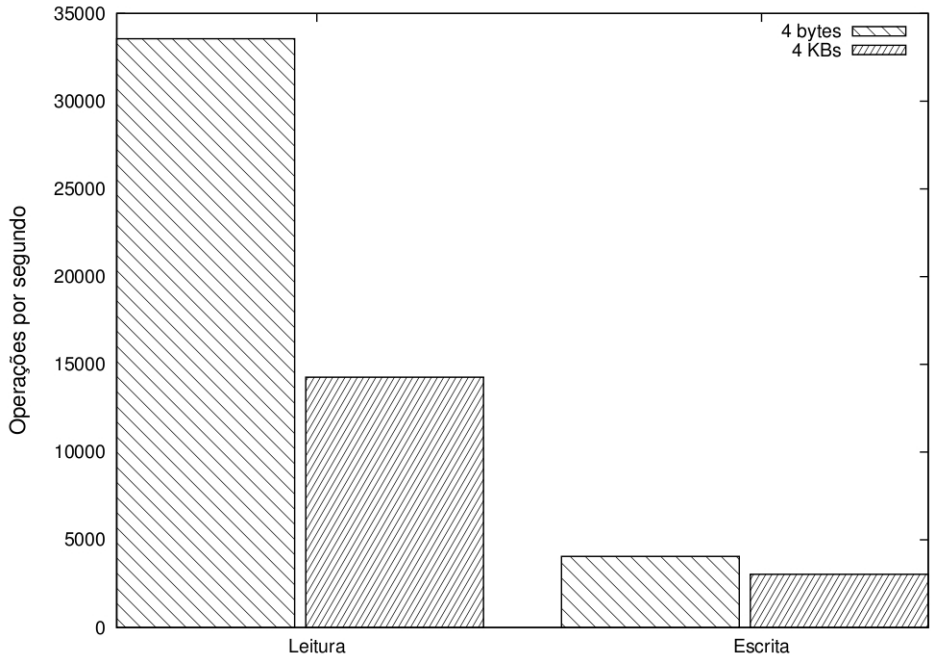


Figura 5.2: Throughput máximo dos protocolos de escrita e leitura.

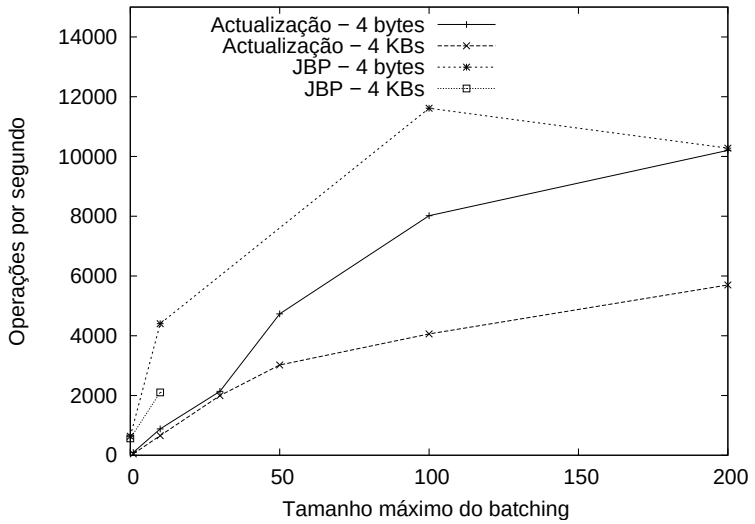


Figura 5.3: Throughput máximo do protocolo de actualização com diferentes tamanhos de batches de mensagens.

*ing* igual a 200, sendo verificado um valor de $10.204$ operações por segundo. Esta constatação é facilmente explicada pois, através do *batching* de mensagens, é possível reduzir o número de instâncias de consenso necessárias para processar os pedidos e, consequentemente, o número de mensagens enviadas e recebidas por cada servidor. Este tipo de resultado está de acordo com outros trabalhos que também exploram esta optimização (e.g., [5, 14, 19]).

Em relação aos valores de *throughput* verificados para o protocolo da máquina de esta-

dos, é interessante fazer a comparação entre o valor máximo registado para o JBP e o valor máximo da actualização (que corre sobre o JBP). Para um objecto de tamanho de 4 bytes, o JBP atinge um *throughput* máximo de $11.614$ operações por segundo com um *batch* com dimensão de 100, ou seja, executa cerca de $14\%$ operações a mais que o protocolo de actualização com batch igual a 200. Tendo em conta o processamento adicional que a actualização tem de efectuar, pode-se concluir que o protocolo de actualização demonstra um desempenho bastante bom em comparação com o valor registado por uma máquina de estados replicada sem realizar processamento.

### 5.3.1 Efeito dos *Multi-Cores*

Adicionalmente, foram realizados testes de carga colocando um dos servidores numa máquina *Xeon quad-core* de 64 bits com processamento 2.3 GHz e 8 GB de RAM operando num ambiente Linux 2.6.21 (todas as máquinas usadas anteriormente como servidores apenas contêm um *core*). Através da medição do número de operações que esse servidor consegue processar, consegue-se perceber o impacto que o uso do modelo de *mulithreading* tem no sistema. A Tabela 5.1 apresenta esses resultados.

| Operação | 1 core | 4 core | Melhoramento (%) |
|---|---|---|---|
| Leitura | 33.557 | 131.579 | 292% |
| Escrita | 4.056 | 10.152 | 150% |
| Actualização sem processamento | 10.204 | 12.305 | 20% |

Tabela 5.1: Número de operações por segundo verificado para objectos de 4 bytes utilizando uma máquina com um *core* e outra com *quad-core*. Para a actualização foi usado um *batching* máximo de 200 mensagens.

Nas operações de leitura e escrita nota-se um aumento elevado do número de operações por segundo efectuadas. Como demonstrado na Tabela 5.1, a operação de leitura apresenta um máximo de $131.579$ operações por segundo num ambiente *quad core*, de tal forma, que é um melhoramento de $292\%$ em relação aos experimentos realizados com um *core*. A escrita também demonstra um aumento considerável chegando a apresentar um melhoramento de $150\%$. Este factos comprovam a utilidade do suporte a *multithreading* para realizar os diferentes passos destes protocolos em paralelo.

Relativamente ao protocolo de actualização, no qual se colocou o servidor *quad core* como líder, não é verificado um aumento de *throughput* tão elevado como constatado na leitura e escrita. Para a realização das actualizações nos testes realizados, a operação $op$, pedida pelo cliente, é bastante simples sendo que não exige processamento quase nenhum. Relembrando que o modelo de *multithreading* na actualização é efectuado no processo de formação da proposta pelo líder (ou seja na realização das várias operações pedidas pelos clientes), só se iria verificar um aumento considerável no caso dessas operações serem custosas para o líder em termos de processamento (e.g. serviço de infraestrutura de chave pública no qual seria necessário realizar constantemente assinaturas sobre objectos).

Deste modo, e apesar dos resultados da actualização não terem aumentado muito, consegue-se mostrar a eficiência que o suporte a *multithreading* oferece ao desempenho geral do sistema, principalmente em ambientes que admitam máquinas com processadores com mais do que um núcleo.

## 5.4 Testes ao *Communication System*

Como referido no capitulo 4, a biblioteca OSC realiza a comunicação com base no *Communication System* (CS) que concretiza *sockets* TCP, estando preparado para enviar e receber mensagens, tanto entre o grupo de servidores, como entre clientes-servidores.

Visto que o desempenho do OSC está altamente dependente do desempenho do CS, foram realizados testes de latência a esta camada, de modo a se perceber o impacto que o uso desta biblioteca tem no sistema. Foram realizados dois tipos de testes: (1.) um experimento em que se envia uma mensagem para um grupo servidores e se espera por $n - f$ mensagens de resposta de servidores diferentes[1] e (2.) outro experimento em que se envia uma mensagem e apenas se espera por uma resposta. Tal como nos testes de latência efectuados ao OSC, também foram realizados 10.000 pedidos por um cliente. Como base de comparação, foi construido um programa de teste simples em Java que apenas abre *sockets* TCP e envia/recebe mensagens adaptando-se aos dois cenários descritos. Os resultados estão apresentados nas Figuras 5.4 e 5.5 sendo importante notar que, em todos estes testes, os objectos são enviados e recebidos.

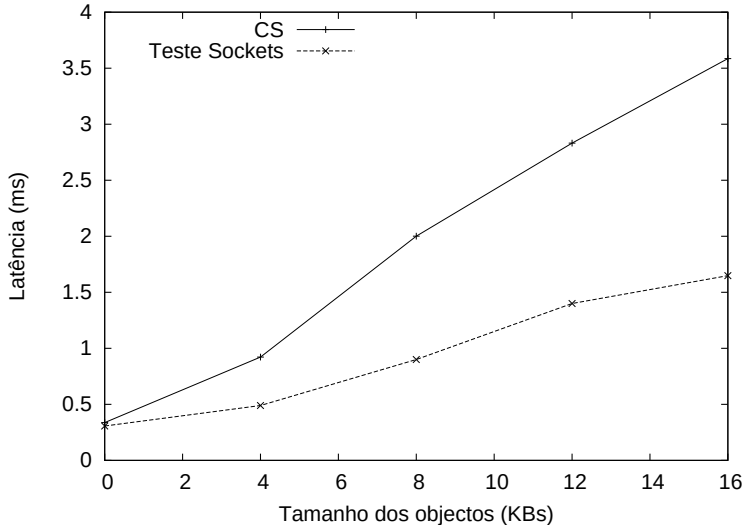


Figura 5.4: Latência do *Communication System* esperando por n - f respostas.

Como se pode reparar nos resultados obtidos, o *Communication System* apresenta valores de latência superiores ao programa de teste concretizado, principalmente quando

[1]Como o ambiente foi configurado para $n = 4$ e $f = 1$, é necessário esperar por 3 respostas.

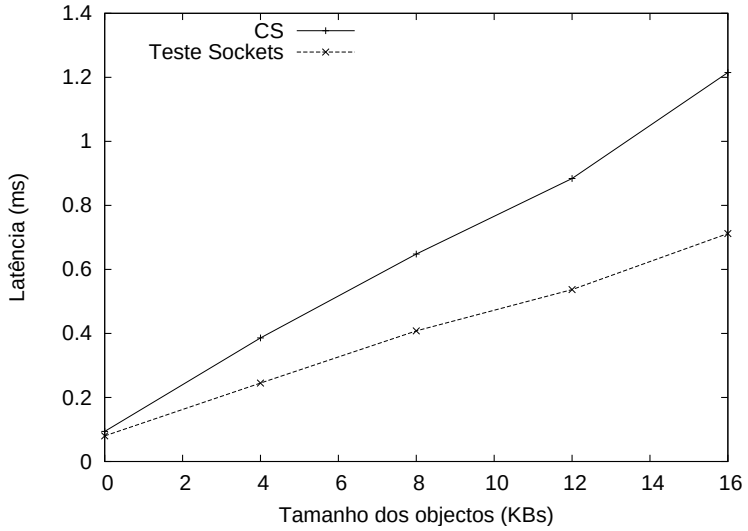


Figura 5.5: Latência do *Communication System* esperando por uma resposta.

espera por $n - f$ respostas de servidores diferentes. O aumento do tamanho dos objectos reflecte-se bastante na latência média do *Communication System*, de tal forma que, entre um objecto de 4 KBs e outro de 16 KBs, a latência sobe 2,5 ms, quando espera por $n - f$ respostas.

Um teste bastante interessante de se realizar, seria a utilização de outra biblioteca de comunicação para o desenvolvimento do OSC, de forma a verificar se é possível melhorar o desempenho. No entanto, por limitações de tempo e de não ser um dos focos deste projecto, este cenário não foi devidamente investigado.

## 5.5 Considerações Finais

Neste capítulo foi efectuada uma avaliação experimental aos protocolos concretizados na biblioteca OSC (leitura, escrita e actualização) assim como uma análise critica aos resultados verificados.

Ao longo do projecto, estes testes permitiram perceber em que passos dos diversos protocolos é detectada um pior desempenho do sistema, de modo a serem realizadas optimizações nessas partes, se possível.

# Capítulo 6

# BFT LDAP

Como referido anteriormente, a biblioteca Objectos de Serviço Confiáveis (OSC) concretiza o SQA, permitindo a construção de aplicações tolerantes a faltas bizantinas. Como demonstra a Figura 6.1, a OSC insere-se como uma camada entre o ambiente e a aplicação fornecendo uma interface de fácil integração.


Figura 6.1: Modelo usado para a concretização de aplicações sobre a OSC.

Para demonstrar a utilização do modelo OSC, na construção de um serviço confiável, foi desenvolvido um LDAP tolerante a faltas bizantinas. Ao longo deste capítulo irá ser abordado a forma como o LDAP foi concretizado, integração com o OSC e testes ao seu desempenho.

## 6.1 Descrição

O LDAP (*Lightweight Directory Access Protocol*) é um protocolo que permite manipular serviços de directório distribuídos [27, 13]. Este protocolo foi construído de forma a possibilitar a interacção com os serviços de directório através de interrogações e modificações [26]. Estes directórios podem ser descritos como base de dados simplificadas contendo conjuntos de objectos organizados de forma lógica e hierárquica. O protocolo LDAP segue o padrão X.500 para organizar os conjuntos de directório e entradas (Figura 6.2) sendo que cada directório é uma árvore de entradas/directórios e as entradas podem conter conjuntos de atributos. Uma entrada pode representar qualquer tipo de objecto, como

por exemplo pessoas, grupos, organizações ou documentos. Um atributo é um par (tipo, valor)[1] que, geralmente, simbolizam características das entradas.

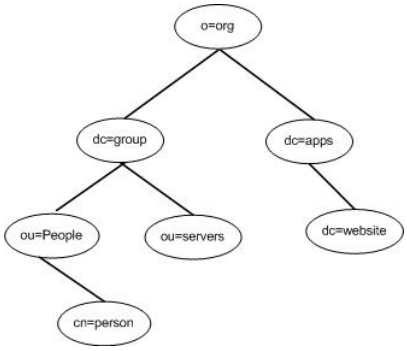


Figura 6.2: Exemplo da estrutura de um directório LDAP.

Tipicamente, o LDAP está assente numa arquitectura cliente-servidor, sendo que o cliente realiza pedidos, e o servidor, que concretiza o serviço de directório, é o responsável por executar os pedidos e enviar a consequente resposta.

Hoje em dia, o LDAP é utilizado por inúmeras aplicações. Exemplos reais podem ir desde aplicações que gerem informação pessoal de uma organização, serviços de translação de nomes ou até mesmo serviços de autenticação.

## 6.2 Operações

Para esta demonstração foi concretizado um LDAP simples que permite que clientes realizem operações sobre um grupo de servidores. Nesta secção são explicadas as operações desenvolvidas.

### 6.2.1 Bind / Unbind

A operação *bind* permite que um utilizador se autentique no serviço LDAP. É necessário que o utilizador se encontre autenticado para conseguir aceder ao serviço e, sobre ele, realizar outras operações. Para isso, é necessário que cada cliente tenha o seu próprio nome assim como a palavra-chave correspondente sendo que é necessário inserir previamente os utilizadores no sistema[2]. Adicionalmente, o *bind* permite também obter uma uma árvore da directoria (ou parte dela) contendo os identificadores únicos dos diversos objectos presentes no sistema.

Por outro lado, a operação *unbind* indica ao serviço a intenção do utilizador de fechar a conexão.

[1] Um atributo é composto por um tipo (e.g., nome, email) e um valor, sendo possível criar novos tipos de atributos.

[2] Visto o objectivo deste LDAP ser exemplificar a integração com a biblioteca OSC não foi construido nenhum sistema de autenticação seguro (com palavras chave cifradas e verificação da integridade das mesmas).

### 6.2.2 Search

A operação *search* permite realizar uma pesquisa sobre o serviço de directórios retornando todas as entradas que correspondem com os parâmetros recebidos. Basicamente, pesquisa os atributos das entradas consoante um certo critério fornecido. Quando encontra correspondência adiciona a entrada em questão ao conjunto solução da operação. Esta operação recebe três parâmetros:

**base:** DN (*Distinguished Name*) que permite especificar a partir de que entrada se realiza a pesquisa;

**scope:** este parâmetro permite definir que elementos por baixo da base se pretende pesquisar. Actualmente o *scope* pode ser três das seguintes variantes:

- base: pesquisa apenas na entrada referida pelo DN.
- one: pesquisa um nível por baixo do DN especificado.
- subtree: pesquisa a entrada especificada juntamente com todo o seu directório, ou seja, todas as entradas por baixo.

**filter:** permite especificar o critério de selecção de entradas a realizar dentro do *scope* definido. O filtro é composto por tipos de atributos, operadores de comparação e valores para os atributos. Devido ao objectivo da construção deste LDAP apenas foram implementados dois tipos de comparação: igual e diferente.

A resposta esperada desta operação deverá ser um conjunto de entradas resultantes da pesquisa ou um conjunto vazio caso não exista no directório referido nenhuma entrada válida.

### 6.2.3 Add

Esta operação permite acrescentar novas entradas no directório. Através do *add* também é possível acrescentar atributos juntamente com a criação da entrada. Para isso, o cliente necessita de fornecer o DN da nova entrada, o DN da entrada que serve de base, e um conjunto de par de atributos (tipo, valor). De notar que não pode existir nenhuma entrada com o mesmo DN pois cada um tem de ser único no sistema.

### 6.2.4 Remove

Esta operação permite apagar entradas do directório. Recebe como argumento o DN da entrada a remover, sendo que, todos os atributos associados a esta entrada também são removidos. A entrada a ser removida não pode conter filhos.

### 6.2.5 Compare

Esta operação permite realizar uma pesquisa a uma entrada fornecida e efectuar uma comparação com os seus atributos. O *compare* recebe um DN e um ou mais pares de atributos (tipo, valor), Esta operação é similar com a operação *search* anteriormente descrita pois também necessita de realizar pesquisas e comparações no directório.

### 6.2.6 Modify

Esta operação permite modificar, adicionar ou remover atributos de uma entrada. Esta é uma operação que recebe como parâmetros um DN correspondente à entrada que se pretende manipular, e um conjunto de pares de atributos (tipo, valor). Caso o atributo já exista na entrada, é realizada uma alteração do seu valor. Caso não exista , é adicionado um novo atributo. Para remover é necessário introduzir o parâmetro 'null' como valor (Exemplo: *modify* entrada mail null).

### 6.2.7 ModifyDN

Esta operação permite modificar o DN de uma entrada, sendo que esta modificação pode significar a alteração de localização da entrada. De notar que todas entradas que se encontrem por baixo da entrada em questão, também sofrem esta modificação. Para operar correctamente é necessário passar como parâmetros: o DN actual da entrada que se quer modificar, o novo DN caso seja para renomear a entrada e ainda o DN onde irá ser inserida a entrada em questão (caso seja para mudar de localização).

## 6.3 Integração

O modelo OSC é uma camada entre o ambiente e a aplicação que resolve questões de consistência e concorrência dando bastante liberdade à aplicação que apenas necessita de lidar com a semântica das operações. Esta secção descreve a integração da aplicação LDAP com o OSC.

### 6.3.1 Cliente

A concretização do cliente LDAP é bastante simples sendo que apenas é necessário transformar as operações LDAP em pedidos SQA, tal como especificado na Tabela 6.1, ficando de seguida à espera do resultado dos seus pedidos.

É importante notar que, as operações tipicamente mais utilizadas neste tipo de sistemas (*search* e *modify*), são concretizadas através de protocolos mais simples (leitura e escrita, respectivamente), que apresentam uma complexidade de mensagens linear e que não requerem hipóteses temporais para o seu correcto funcionamento.

| Operação | Protocolo SQA |
|---|---|
| *Bind* | Escrita e Leitura |
| *Unbind* | Escrita |
| *Search* | Leitura |
| *Modify* | Escrita |
| *Add* | Actualização |
| *Remove* | Actualização |
| *ModifyDN* | Actualização |

Tabela 6.1: Operações concretizadas pelo LDAP e respectivos protocolos SQA.

Em relação à operação *Bind*, é necessário efectuar uma escrita e uma leitura para realizar correctamente a operação. A escrita é usada para autenticar o utilizador enquanto que a leitura tem o objectivo de obter como resultado uma estrutura de parte da árvore de directório com os identificadores únicos dos objectos. Na realidade, o cliente apenas efectua uma operação de escrita para se autenticar. No lado do servidor, este, após realizar a autenticação, efectua a leitura (sem ser necessário o envio do pedido pelo cliente) respondendo com o resultado.

### 6.3.2 Servidor

Ao se desenvolver o servidor LDAP foi necessário definir o modelo de objectos a seguir, sendo que qualquer objecto manipulável deverá estender a classe `AQSObject` e implementar os métodos responsáveis por executar as operações de leitura, escrita e actualização. Por este motivo, definiu-se que as entradas (directorias) e os atributos são derivados do `AQSObject` sendo que, por isso, possuem a sua própria estampilha temporal e certificado que permite verificar a validade dos objectos. A Figura 6.3 ilustra este modelo de objectos.

Para realizar a autenticação dos utilizadores no sistema foi desenvolvida uma classe denominada `LDAPUser` com uma variável de estado que permite que os servidores alterem e verifiquem o estado do utilizador. Estes objectos, apesar de pertencerem à classe `AQSObject` não são visíveis nem manipuláveis pelos utilizadores.

Todas as operações LDAP necessitam de recorrer à funcionalidade de operações em múltiplos objectos de modo a serem concretizadas correctamente. O OSC dá suporte para se realizar este tipo de operações mas é importante saber que é da responsabilidade da aplicação concretiza-las correctamente.

Relativamente às operações que envolvem criar ou modificar objectos (e.g., *add*), elas são concretizadas como actualizações que causam a criação de um novo objecto (através da invocação de métodos do `AQSObjectManager`). É importante constatar que é garantido a formação de um nova estampilha temporal e certificado de modo a ser possível verificar a validade do estado actual do objecto.

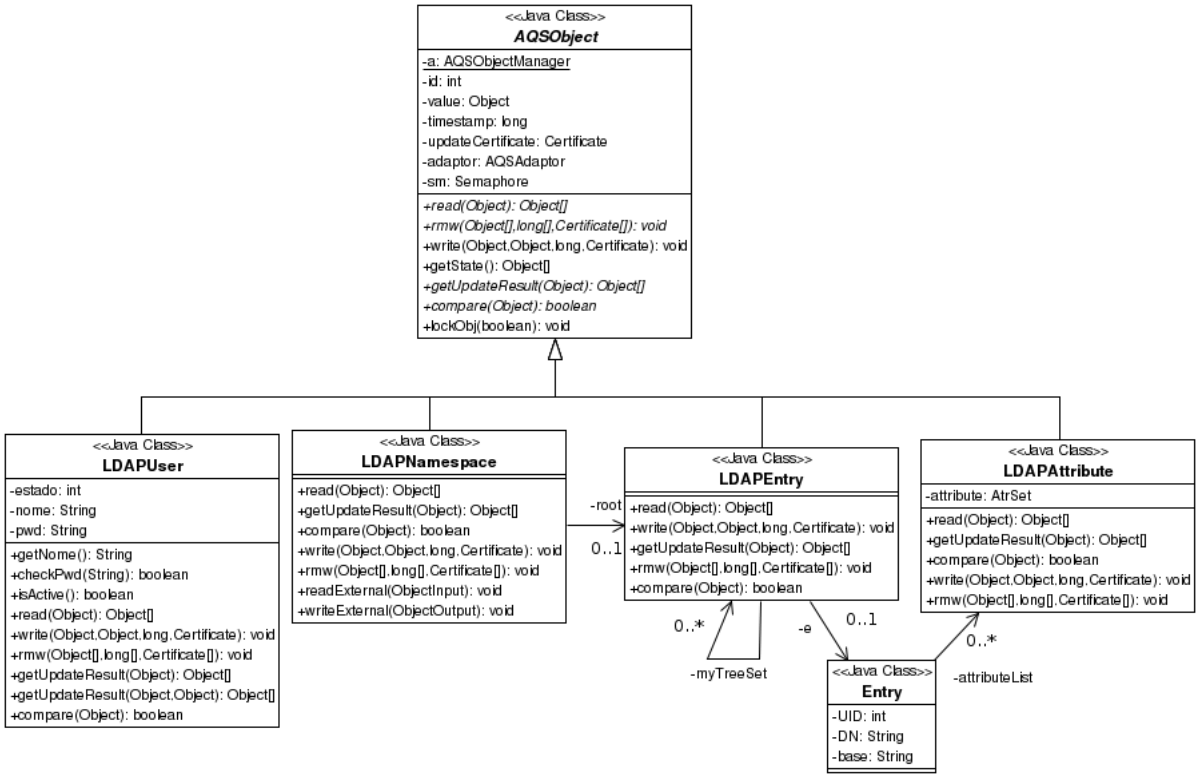


Figura 6.3: Modelo de classes do LDAP.

## 6.4 Avaliação

A maior contribuição do LDAP concretizado é a de demonstrar que facilmente se desenvolve qualquer tipo de sistemas sobre a biblioteca OSC. Por isso, muitas optimizações que, tipicamente, são concretizadas num LDAP, conforme [26], não foram desenvolvidas (e.g., *cache* do lado do cliente). No entanto, já é possível obter alguns resultados preliminares sobre o desempenho do serviço LDAP tolerante a faltas bizantinas.

Para a realização destes testes foram escolhidas três operações comummente executadas que fazem uso dos três protocolos básicos suportados pela biblioteca OSC. Estas operações foram executadas 10.000 vezes sem contenção e a latência de acesso média das operações foi calculada, e reportada nas Figuras 6.4 e 6.5.

Conforme esperado, o resultado das operações reflecte o custo dos protocolos subjacentes por elas usado: a operação *modify* apresenta um desempenho melhor que o *add* pois este último requer o uso do protocolo de actualização, muito mais pesado que o protocolo de escrita. Para objectos de 4 KBs, a diferença entre a latência obeservada chega a quase 4 ms. Em relação ao *search*, os resultados variam consoante o número de objectos de 4 KBs retornados, e reflectem exactamente o custo da operação de leitura observado na Figura 6.5, apresentando uma latência de 2,5 ms a efectuar esta operação retornando um único objecto.

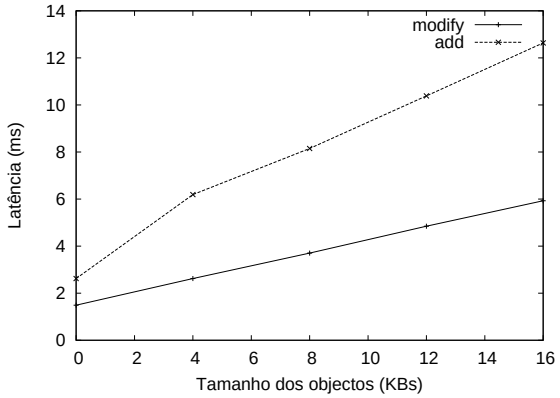


Figura 6.4: Latência das operações *add* e *modify*.

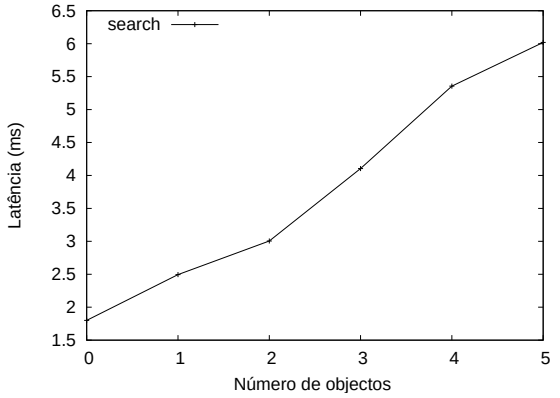


Figura 6.5: Latência da operação *search* com diferentes números de objectos de 4KB retornados.

## 6.5 Considerações Finais

Este capítulo serviu para exemplificar a construção de um serviço confiável concretizado sobre a biblioteca OSC. Através da construção deste LDAP foi possível ganhar uma percepção da forma de como as aplicações têm de ser integradas, de modo a conseguir fornecer um serviço seguro.

Através dos testes realizados foi comprovado que é possível construir este tipo de aplicação tolerantes a faltas bizantinas sem que o seu desempenho fique gravemente prejudicado. É importante notar que os resultados foram obtidos para uma versão preliminar do serviço LDAP que poderia ainda ser melhorado e optimizado de modo a aumentar a sua eficiência.

# Capítulo 7

# Conclusão

Neste tese foi apresentado um novo modelo de replicação tolerante a faltas bizantinas chamado Sistemas de Quoruns Activos, que foi inicialmente elaborado pelo meu orientador de projecto, o Professor Alysson Neves Bessani. Esta nova proposta tem a particularidade de ser o primeiro modelo a juntar sistemas de quoruns bizantinos com máquina de estados replicada de forma a permitir realizar diferentes operações usando os protocolos mais simples necessários para satisfazer as semânticas requeridas. Este modelo faz a distinção clara entre protocolos de escrita e de actualização, sendo que, este facto contribui para um melhor desenvolvimento de futuras aplicações que operem sobre o SQA. Deste modo, é possível que as aplicações realizem todas as suas operações recorrendo aos protocolos concretizados. Uma característica vantajosa do SQA é ser um serviço de replicação não determinista, sendo que é um dos primeiros sistemas a apresentar esta característica sem ser necessário usar mecanismos adicionais.

O objectivo principal deste projecto era desenvolver uma biblioteca que concretizá-se a proposta SQA. Com essa finalidade, foi desenvolvida a biblioteca Objectos de Serviço Confiáveis (OSC) baseado no modelo apresentado formando, deste modo, uma camada que deve ser inserida entre o ambiente e a aplicação. Esta camada permite a construção de aplicações tolerantes a faltas bizantinas, dando possibilidade ao programador de se abstrair desses conceitos. Esta biblioteca, permite a integração com aplicações de forma fácil e simples sendo que a única dificuldade será ao nível da semântica das operações e converter essas operações especificas em protocolos SQA. O OSC apresenta características bastante interessantes como o suporte ao *multithreading* que, nos dias que correm, vai ser uma técnica cada vez mais usada. Outra característica bastante interessante é o suporte a operações em múltiplos objectos que dá a possibilidade de, com um só pedido, uma operação afectar mais do que um objecto.

Relativamente à fase de testes da biblioteca OSC, é importante notar que esta fase foi realizada várias vezes ao longo do projecto. Nos primeiros meses de projecto, foi desenvolvida uma versão preliminar da OSC, sem optimizações nenhumas, servindo essa versão para uma primeira análise do sistema. Após obtenção dos primeiros resultados foi

possível ter uma análise critica e, deste modo, pensar e concretizar certas optimizações que se revelaram cruciais para melhorar o desempenho geral dos protocolos. De notar que, nesta fase a ajuda e intervenção do orientador foi essencial para o correcto desenvolvimento das optimizações. A maior parte dessas optimizações foram baseadas em outros protocolos de replicação tolerantes a faltas bizantinas já existentes. No entanto, a biblioteca OSC introduziu uma optimização interessante, não verificada noutras propostas, que permite realizar operações de actualização usando um protocolo de difusão com ordem total sem que seja necessário enviar os estados dos objectos em questão. Os resultados obtidos nesta versão final do projecto mostram que a OSC apresenta um desempenho razoável relativamente à abordagem tradicional da máquina de estados replicada.

Para demonstrar a construção de serviços confiáveis que recorram à biblioteca desenvolvida, foi elaborado um serviço LDAP tolerante a faltas bizantinas. Este LDAP é uma versão bastante simples mas exemplifica a forma como uma aplicação deve ser construída sobre a biblioteca OSC. Com o mesmo objectivo foram obtidos valores de latência de modo a se provar que é possível realizar aplicações confiáveis sem que o seu desempenho seja gravemente prejudicado.

Ao longo deste projecto foram aprendidos inúmeros conceitos bastante interessantes nesta linha de investigação que junta a segurança com a confiabilidade. A nível pessoal, foi bastante gratificante e entusiasmante ter a oportunidade de trabalhar numa área que me desperta interesse elevado. Outro aspecto que foi bastante motivante, é o facto desta linha de investigação estar permanentemente em evolução nos últimos anos e que, apesar de no mundo real muitas empresas e companhias ainda não estarem sensibilizadas para a necessidade dos vários paradigmas de tolerância, num futuro próximo esta área irá ser o foco de muitas organizações.

## 7.1 Trabalho Futuro

De acordo com os estudos realizados e aos resultados obtidos torna-se claro que este projecto serviu de motivação para que outras ideias sejam aplicadas por cima do modelo elaborado. Ao longo do decorrer do projecto, surgiram várias ideias que, por motivos de tempo e de não ser o real objectivo do projecto, não foram desenvolvidas. Esta secção descreve algumas dessas ideias.

**Balanceamento de carga:** No protocolo de actualização do SQA, é assumido a presença de um líder que é o responsável por executar as operações, disseminando os resultados pelos restantes servidores. Facilmente se percebe que o líder do grupo tem uma carga bastante superior aos outros servidores sendo um ponto de estrangulamento do sistema. De modo a colmatar esta anomalia, pode ser desenvolvido um modelo de balanceamento de carga, no qual cada conjunto de objectos tem um líder

que pode ser qualquer um dos $n$ servidores. Caso não houvesse faltas, pode fazer-se uma estimativa de que o sistema iria ter a capacidade de executar $n$ vezes mais operações do que o normal.

**Operações que invoquem mais do que um protocolo SQA:** Certas aplicações concretizam operações que, em termos de semântica, requerem mais do que uma invocação de um protocolo SQA (e.g., a operação *bind* do serviço LDAP desenvolvido). Uma ideia interessante seria concretizar um certo modelo ou opção que permita que um pedido de uma aplicação cliente execute múltiplas operações SQA no sistema (e.g., executar uma escrita seguida de leitura). Neste caso, ao se concretizar esta extensão da biblioteca OSC seria necessário ter em atenção a não violação da especificação dos vários protocolos do sistema.

**Melhorar o sistema de comunicação:** Ao longo do projecto, foi constatado que a biblioteca usada para realizar a comunicação, *Communication System*, poderia ser melhorada visto que havia certas opções de implementação que não favoreciam o desempenho geral dos protocolos do SQA. Seria vantajoso para o SQA se, por exemplo, a comunicação entre o grupo de servidores fosse processada de forma independente da comunicação entre os clientes e servidores (cenário que não acontece com o sistema de comunicação actual). Por motivos de tempo, não foi verificado se, com um sistema de comunicação melhor, é possível obter melhores resultados de latência e de *throughput* para o SQA.

**Avaliação:** Nesta tese, os testes e a avaliação realizada à biblioteca OSC apenas consideravam cenários óptimos, nos quais não havia concorrência e não se verificavam falhas dos servidores, nem dos clientes. Seria bastante útil para o estudo do SQA se fossem realizados testes em cenários adversos ao sistema. Relativamente à concorrência poderiam ser efectuados experimentos em que houvesse diversos pedidos concorrentes a um objecto, e outros cenários em que se criá-se concorrência entre diferentes protocolos (e.g., escritas com actualizações). Para se avaliar o comportamento do sistema perante faltas poderia simular-se diversos casos, como por exemplo, clientes que enviam pedidos para metade do servidores, servidores que enviam valores errados, líder que não propõe ordem para mensagens, ou que envia diferentes ordens para a mesma mensagem. Adicionalmente, poderia verificar-se o comportamento do SQA para sistemas com $n > 4$ servidores (e.g., $n = 7$, $n = 10$).

**Melhorar LDAP:** Como referido anteriormente, poderiam ser desenvolvidas várias optimizações para o serviço LDAP concretizado, de modo a melhorar o seu desempenho [26]. Deste forma, seria possível demonstrar de forma mais clara o bom desempenho de aplicações mesmo quando seguem o paradigma da tolerância a faltas bizantinas.

**Outras aplicações:** Poderia concretizar-se mais aplicações sobre a biblioteca OSC e compara-las com outras aplicações do mesmo tipo, mas que usem abordagens diferentes. Por exemplo, a construção de um espaço de tuplos sobre o SQA poderia ser comparado ao *DepSpace* (que usa máquina de estados) [2].

# Bibliografia


[1] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 59–74, New York, NY, USA, 2005. ACM Press.

[2] A. N. Bessani, E. P. Alchieri, M. Correia, and J. S. Fraga. Depspace: a Byzantine fault-tolerant coordination service. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 163–176, New York, NY, USA, 2008. ACM.

[3] A. N. Bessani, M. Correia, J. da Silva Fraga, and L. C. Lung. An efficient Byzantine-resilient tuple space. *IEEE Transactions on Computers*, 58(8):1080–1094, 2009.

[4] A. N. Bessani, M. Correia, and P. Sousa. Active quorum systems: Specification and correctness proof. Technical report, Dep. of Informatics, University of Lisbon, Dec. 2008. Available at `http://www.di.fc.ul.pt/~bessani/aqs-tr.pdf`.

[5] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.

[6] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

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

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

[9] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: a hybrid quorum protocol for Byzantine fault tolerance. In *OSDI '06: Proceedings*

*of the 7th symposium on Operating systems design and implementation*, pages 177–190, Berkeley, CA, USA, 2006. USENIX Association.

[10] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35:288–323, 1988.

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

[12] D. Gifford. Weighted voting for replicated data. In *Proc. of the 7th ACM Symposium on Operating Systems Principles*, pages 150–162, Dec. 1979.

[13] J. Hodges and R. Morgan. Lightweight Directory Access Protocol (v3): Technical Specification. RFC 3377 (Proposed Standard), Sept. 2002.

[14] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative Byzantine fault tolerance. *SIGOPS Oper. Syst. Rev.*, 41(6):45–58, 2007.

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

[16] B. Liskov and R. Rodrigues. 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, 2006. IEEE Computer Society.

[17] D. Malkhi and M. Reiter. Byzantine quorum systems. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 569–578, New York, NY, USA, 1997. ACM.

[18] J.-P. Martin and L. Alvisi. Fast Byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, 2006.

[19] H. Moniz, N. F. Neves, M. Correia, and P. Verissimo. Randomized intrusion-tolerant asynchronous services. *Dependable Systems and Networks, International Conference on*, 0:568–577, 2006.

[20] N. F. Neves. Tolerância a intrusões em sistemas informáticos. DI/FCUL TR 05–7, Department of Informatics, University of Lisbon, May 2005. Prémio IBM de Ciência 2005.

[21] R. R. Obelheiro, A. N. Bessani, L. C. Lung, and M. Correia. How practical are intrusion-tolerant distributed systems? DI-FCUL TR 06–15, Dep. of Informatics, Univ. of Lisbon, Sept. 2006.

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

[23] P. Verissimo, N. F. Neves, and M. P. Correia. Intrusion-tolerant architectures: Concepts and design. In *Architecting Dependable Systems*, volume 2677 of *LNCS*. 2003.

[24] G. Veronese, M. Correia, A. Bessani, L. C. Lung, and P. Verissimo. Minimal Byzantine fault tolerance: Algorithms and evaluation. DI-FCUL TR 09-15, Departamento de Informática, Universidade de Lisboa, jul 2009.

[25] P. Veríssimo. Intrusion tolerance: Concepts and design principles. a tutorial. DI/FCUL TR 02–6, Department of Informatics, University of Lisbon, July 2002.

[26] X. Wang, H. Schulzrinne, D. Kandlur, and D. Verma. Measurement and analysis of ldap performance. *IEEE/ACM Trans. Netw.*, 16(1):232–243, 2008.

[27] W. Yeong, T. Howes, and S. Kille. Lightweight Directory Access Protocol. RFC 1777 (Historic), Mar. 1995.

[28] P. Zielinski. Paxos at war. Technical Report UCAM-CL-TR-593, University of Cambridge Computer Laboratory, Cambridge, UK, June 2004.