João Vaz

Licenciado em Engenharia Informática

Tolerância a Faltas Bizantinas em Servidores RMI

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientador : Nuno Preguiça, Prof. Auxilar , Universidade Nova de Lisboa

Co-orientador : João M. Lourenço, Prof. Auxilar , Universidade Nova de Lisboa

Júri:

Presidente:	Prof. Doutora Carla Ferreira
Arguente:	Prof. Doutor Alysson Bessani
Vogal:	Prof. Doutor Nuno Preguiça

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Novembro, 2011

Tolerância a Faltas Bizantinas em Servidores RMI

Copyright © João Vaz, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa

A Faculdade de Ciências e Tecnologia e a Universidade Nova de Lisboa têm o direito, perpétuo e sem limites geográficos, de arquivar e publicar esta dissertação através de exemplares impressos reproduzidos em papel ou de forma digital, ou por qualquer outro meio conhecido ou que venha a ser inventado, e de a divulgar através de repositórios científicos e de admitir a sua cópia e distribuição com objectivos educacionais ou de investigação, não comerciais, desde que seja dado crédito ao autor e editor.

Para Luís Armando Vaz

Agradecimentos

Li algures que a melhor parte de fazer uma dissertação é escrever a secção de agradecimentos. Confirmo.

Em primeiro lugar gostaria de agradecer aos meus orientadores, sem os quais nunca teria sido possível elaborar este trabalho, Nuno Preguiça e João Lourenço, pela oportunidade que foi trabalhar com ambos. Devo-lhes, para além desta dissertação e da sua orientação, em especial quando andei mais perdido, algumas lições importantes para a minha formação académica e pessoal. Quero também agradecer ao Rui Garcia a ajuda na solução de alguns problemas na fase inicial deste projecto.

Em segundo lugar quero agradecer a todos os meus amigos que, por coincidência, também foram meus colegas de curso nestes últimos anos. Em particular, pelas inúmeras experiências que partilhámos na sala de aula e fora dela, e também por aquelas que iremos partilhar daqui para a frente, ao Ricardo Alves, Vasco Pessanha, Pedro Tempera, Filipe Firmo, Rui Azevedo, Joana Dias e Diogo Duarte.

É importante reconhecer que sem a minha família nunca teria chegado aqui. Por todo o apoio ao longo dos anos nos bons momentos e, especialmente, nos maus. Um enorme Obrigado à minha mãe, Maria Emília Vaz, à minha irmã, Ana Rita Vaz, e em especial, ao meu pai, Luís Armando Vaz. Gostaria ainda de agradecer à minha avó, Emília Almeida (Emilinha), pelo gigantesco exemplo de força e determinação que sempre representou para mim e, estou certo, para todos nós.

Por último quero agradecer à minha namorada, Ana Luísa Mariano, que partilhou comigo os bons momentos e me motivou para ultrapassar aqueles menos bons. Sem ela, nem este trabalho estaria terminado nem eu seria metade da pessoa que sou. Obrigado!

O trabalho apresentado nesta dissertação foi suportado pela Fundação para a Ciência e Tecnologia (FCT/MCTES), no contexto do projecto Byzantium (PTDC/EIA/74325/2006) e do projecto RepComp (PTDC/EIA-EIA/108963/2008).

Resumo

Os avanços registados recentemente ao nível do hardware tornaram possível melhorar a performance e eficiência da aplicações em geral através da computação paralela. Com este aumento dos recursos disponíveis, os sistemas computacionais evoluíram no sentido de exigir uma maior fiabilidade, disponibilidade e tolerância a falhas arbitrárias (bizantinas).

Um componente que exiba comportamentos bizantinos, continua a responder aos pedidos, mas a produzir valores incorrectos. A detecção deste tipo de falhas é um processo complexo, uma vez que estas podem permanecer dormentes durante longos períodos de tempo. A solução passa por implementar algoritmos de tolerância a falhas bizantinas (BFT) robustos, baseados em replicação e em protocolos de consenso que, no entanto, têm um impacto negativo no desempenho. De entre as técnicas que podemos aplicar para limitar esta perda, existem duas que queremos salientar: a introdução de operações concorrentes nos servidores e a utilização de execução especulativa. O mecanismo de invocações remotas da linguagem Java (RMI) permite transportar para as aplicações distribuídas o modelo de programação das aplicações não distribuí- das. Este mecanismo é suportado por uma arquitectura cliente/servidor que, apesar de apresentar uma boa performance, torna estas aplicações pouco tolerantes a falhas. Assim, o nosso objectivo é introduzir tolerância a falhas bizantinas em aplicações RMI, através de um mecanismo de replicação implícito.

Para testar o nosso trabalho, efectuámos testes utilizando o JNFS, um sistema de ficheiros distribuído implementado sobre RMI. Os resultados permitem concluir que o uso de execução especulativa minimiza o impacto dos algoritmos de tolerância a falhas bizantinas.

Abstract

The advances in computer hardware made it possible to improve the performance and efficiency of general applications through parallel computing. Such powerful hardware allows enterprise systems to evolve and provide better guarantees to users, like reliability, availability and bizantine fault-tolerance.

A component that exhibitis byzantine behaviour, continues to produce incorrect values while responding to every request. This fact makes it harder for distributed systems to endure byzantine failures, since they may remain undetected for a long period. In order to overcome such failures, byzantine fault tolerant systems rely on replication with strong agreement protocols, which harm performance and throughput. Among the techniques we may use to overcome these limitations there are: introduction of concurrent execution in the servers and speculative execution.

Java's remote method invocation (RMI) allows seamless remote method invocation on objects in different virtual machines. This mechanism is suported by a client/server architecture which, even though it presents good performance, does not help distributed applications to be reliable and fault-tolerant. Hence, our goal is to create byzantine fault-tolerant applications, through an implicit replication mechanism.

In order to validate our work, we performed some tests using JNFS, a distributed file system implemented over RMI. The results allow us to conclude that the use of speculative execution minimizes the overhead generetad by bizantine fault-tolerant systems.

Conteúdo

1 Introdução **1**
1.1 Contexto 1
1.2 Problema 2
1.3 Aproximação 3
1.4 Contribuições 3
1.5 Estrutura do Documento 3

2 Trabalho Relacionado **5**
2.1 RMI 5
2.1.1 Arquitectura 6
2.1.2 Outras Propostas 7
2.2 Replicação 8
2.2.1 Características 9
2.2.2 Coordenação 11
2.2.3 Tolerância a falhas bizantinas 11
2.2.4 Byzantium 13
2.3 Especulação 15
2.3.1 Especulação ao nível das Threads 16
2.3.2 Especulação ao nível do Sistema 18
2.3.3 Especulação em sistemas de tolerância a falhas bizantinas 19

3 Arquitectura **23**
3.1 Desenho Geral 24
3.2 Estrutura da Solução 24
3.2.1 Especulação baseada em futuros 25
3.2.2 Replicação com tolerância a falhas bizantinas 25
3.2.3 Execução concorrente no servidor 26
3.3 Caso experimental 29

4 Implementação **31**
4.1 RMI . . . 31
4.1.1 Protocolo RMI . . . 34
4.2 RMI sobre UDP . . . 35
4.3 RMI com tolerância a falhas bizantinas . . . 38
4.3.1 Biblioteca BFT . . . 38
4.3.2 Implementação . . . 42
4.4 Mecanismo de Locks . . . 45

5 Validação Experimental **49**
5.1 Metodologia . . . 49
5.2 Benchmark . . . 50
5.2.1 Resultados . . . 50
5.3 Especulação entre operações independentes . . . 53
5.3.1 Resultados . . . 54
5.4 Especulação entre operações dependentes . . . 55
5.4.1 Resultados . . . 56
5.5 Comparação . . . 58
5.6 Execução Concorrente no Servidor . . . 58

6 Conclusão **61**
6.1 Trabalho Futuro . . . 62
6.1.1 Comparação de Resultados . . . 62
6.1.2 Modelo de Execução Especulativa . . . 62
6.1.3 Aplicação a outros modelos de falhas . . . 62
6.1.4 Utilização de memória transaccional . . . 62
6.1.5 Mecanismo de Locks automático . . . 62

Lista de Figuras

2.1 Arquitectura do RMI. 7
2.2 Arquitectura do Aroma.(Retirado de [NMMS00]) 8
2.3 Execução normal do algoritmo PBFT. (Extraído de [CL02]) 13
2.4 Arquitectura do Byzantium. 14
2.5 Exemplo de execução especulativa. (a) mostra a execução normal, sem especulação. (b) mostra a forma como a especulação pode reduzir o tempo de espera. (Retirado de [CG99]) 17
2.6 Speculator. (Retirado de [NCF06].) 19
2.7 Execução normal do algoritmo PBFT-CS. (Extraído de [WCN+09]) 20
2.8 Execução normal do algoritmo Zyzzyva. (Extraído de [KAD+07]) 21

3.1 Exemplo de especulação baseada em futuros. (Extraído de [Cou]) 25
3.2 Arquitectura de uma aplicação RMI, utilizando a biblioteca BFT. 26
3.3 Interface do mecanismo de gestão de recursos. 27
3.4 Exemplo de um método onde é utilizado o `ContentionManager`. 28
3.5 Classe abstracta Lock. 29
3.6 Arquitectura do JNFS. Retirado de [Rad97]. 29
3.7 Arquitectura do JNFS, no caso de ser utilizado BFT. 30

4.1 Exemplo de um *socket* RMI. 32
4.2 Exemplo de uma fábrica de *sockets* do cliente. 33
4.3 Esquema de uma conexão RMI. 33
4.4 Composição da mensagem de uma invocação RMI. 34
4.5 Tipos de mensagens RMI. 35
4.6 Exemplo de comunicação entre dois objectos RMI, utilizando `UdpSocket`. 36
4.7 Excerto da classe `UdpOutputStream`. 37
4.8 Excerto da classe `Listener`. 38
4.9 Excerto da classe `Listener`. 39
4.10 Arquitectura de uma aplicação, utilizando a biblioteca BFT. 40

4.11 Arquitectura de uma aplicação RMI, utilizando a biblioteca BFT. 43
4.12 Excerto da classe `BFTOutputStream`. 43
4.13 Excerto da classe `Listener`. 44
4.14 Exemplo de um método onde é utilizado o `ContentionManager`. 46
4.15 Exemplificação da forma como o mecanismo de contenção funciona. . . . 47
4.16 Exemplo de um método onde é utilizado o `ContentionManager` no âmbito do JNFS. 48

5.1 Fase 2 do *benchmark*. 51
5.2 Comparação dos tempos obtidos nas três versões. 52
5.3 Comparação das operações por segundo obtidas nas três versões. 52
5.4 Operações efectuadas na fase 2 do *benchmark*, com especulação. 53
5.5 Comparação dos tempos obtidos nas três versões. 54
5.6 Comparação das operações por segundo obtidas nas três versões. 55
5.7 Operações efectuadas na fase 2 do *benchmark*, com especulação. 56
5.8 Comparação dos tempos obtidos nas três versões. 57
5.9 Comparação das operações por segundo obtidas nas três versões. 57
5.10 Comparação do número de operações por segundo utilizando TCP. 58
5.11 Comparação do número de operações por segundo utilizando UDP. 58
5.12 Comparação do número de operações por segundo utilizando BFT. 59
5.13 Comparação do número de operações por segundo entre as duas bibliotecas sem especulação. 60
5.14 Comparação do número de operações por segundo entre as duas bibliotecas utilizando especulação entre operações independentes. 60
5.15 Comparação do número de operações por segundo entre as duas bibliotecas utilizando especulação entre operações dependentes. 60

Lista de Tabelas

Listings

Introdução

1.1 Contexto

Os avanços dos últimos anos no hardware dos computadores tornam possível implementar técnicas com o objectivo de melhorar a performance e eficiência das aplicações em geral, em especial através de computação paralela.

Componentes com tanto poder computacional, como processadores multi-core, também levam a um enorme desperdício de recursos uma vez que, a maior parte do tempo, as aplicações não necessitam dos mesmos. A execução especulativa é uma técnica que permite aproveitar estes recursos para melhorar a performance das aplicações.

A crescente procura por serviços altamente disponíveis e fiáveis leva a que sejam cada vez mais comuns os sistemas distribuídos com elevadas capacidades de tolerância a falhas. A maioria destes sistemas apenas admite falhas normais, normalmente conhecidas como *fail-stop*, em que um componente ao falhar pára totalmente a sua execução, deixando de estar disponível. A razão para que isto aconteça é o facto de este tipo de falhas ser relativamente simples de detectar. Por outro lado, ao admitirmos comportamentos arbitrários, ou bizantinos, uma aplicação pode falhar de várias formas, continuando a produzir resultados em resposta aos pedidos. Assim, este tipo de falhas pode permanecer indetectável durante longos períodos de tempo e o processo da sua detecção é bastante complexo. Desta forma, de modo a não comprometer a sua performance, a maioria dos sistemas não tolera comportamentos bizantinos.

No entanto, a drástica descida nos preços do hardware torna possível a construção de sistemas mais complexos e com maiores requisitos computacionais. Sistemas tolerantes a falhas bizantinas (BFT - Bizantine Fault Tolerant), muitas vezes necessitam de passar por um processo de detecção de falhas através da troca de mensagens entre os vários

servidores que compõem os sistemas, o que prejudica bastante a performance [CL02, KAD+07].

O mecanismo de invocações remotas da linguagem Java (RMI) permite transportar para as aplicações distribuídas o modelo de programação das aplicações normais. Este mecanismo é suportado por uma arquitectura cliente/servidor que, apesar de apresentar uma boa performance, torna estas aplicações pouco robustas no que toca a tolerância a falhas.

Uma vez que o RMI torna transparentes para o cliente os detalhes da implementação do servidor, é interessante encontrar mecanismos de replicação que permitam tornar estas aplicações mais robustas, mantendo a semântica de comunicação intacta. Em particular, as aplicações necessitam de mecanismos de replicação quando o objectivo é suportar falhas bizantinas. Estes mecanismos devem comportar-se como se tratasse de uma máquina de estados replicada [Sch90].

A grande desvantagem destes sistemas está na clara perda de performance que sofrem, devido ao elevado número de mensagens que necessitam de ser trocadas de modo a garantir o cumprimento do protocolo. A execução especulativa pode ser considerada uma técnica que permite mascarar a latência dos sistemas, que consiste na previsão do resultado final de uma operação longa de forma a ser possível antecipar a execução das operações seguintes. Considera-se que estas operações são executadas de forma *especulativa*. No caso de a previsão estar correcta, terá sido poupado o tempo de espera pelo resultado final da operação. Caso contrário, teremos de recuperar o estado da aplicação anterior à especulação e voltar a executar todas as operações utilizando o resultado correcto. Esta técnica pode trazer altos ganhos de performance. No entanto, é necessário fazer uma escolha cuidada e ponderada sobre quais as operações a executar desta forma, uma vez que os erros na previsão podem levar a uma ainda maior perca de performance.

Um dos aspectos importantes da execução especulativa é que devemos sempre garantir que resultados incorrectos provenientes de uma especulação errada nunca produzirão estados inconsistentes. Desta forma, apenas devemos tornar permanentes dados especulativos na altura em que obtivermos a confirmação do seu resultado final.

1.2 Problema

As aplicações distribuídas baseadas no RMI do Java são suportadas por uma arquitectura cliente/servidor. Este tipo de arquitecturas é bastante frágil em termos de fiabilidade e os sistemas distribuídos actuais necessitam cada vez mais de fornecer serviços fiáveis.

Uma das formas de melhorar a fiabilidade de um sistema é através de mecanismos de tolerância a falhas. Estes, geralmente, são baseados em técnicas de replicação, tanto de serviços como de dados o que, potencialmente, prejudica a performance. No caso de admitirmos um comportamento bizantino por parte dos componentes do sistema, a performance sai ainda mais deteriorada, uma vez que as operações efectuadas pelos clientes têm de ser ordenadas de forma total.

A replicação de objectos remotos no Java não é uma tarefa trivial, uma vez que a arquitectura das aplicações é radicalmente alterada cabendo ao programador a definição, de forma explícita, da estrutura e do protocolo de comunicação de todos os componentes do sistema.

Este impacto negativo pode ser minimizado através do uso de execução especulativa. Desta forma, em vez de esperar pelo resultado final de uma operação longa, o cliente pode executar as operações seguintes, poupando o tempo que estaria em espera. No entanto, é necessário minimizar o número de operações que têm de ser executadas por utilizar valores especulativos errados.

1.3 Aproximação

A nossa aproximação passa por introduzir um mecanismo de replicação e tolerância a falhas bizantinas implícito. Assim, as aplicações poderão beneficiar das vantagens deste mecanismo, mantendo a semântica de comunicação, sem que necessitem sofrer qualquer alteração.

Iremos também utilizar mecanismos de especulação como forma de reduzir o impacto negativo em termos de performance que os algoritmos de tolerância a falhas bizantinas trazem às aplicações.

1.4 Contribuições

As contribuições apresentadas neste trabalho são:

- **Replicação implícita no RMI** – É apresentado um mecanismo de replicação implícita de objectos remotos. Este mecanismo é totalmente transparente para as aplicações e permite que sejam criados vários servidores em máquinas virtuais diferentes que comunicam entre si.

- **Suporte para tolerância a falhas bizantinas no RMI** – Através do mecanismo de consenso entre as réplicas, as aplicações passam a tolerar comportamentos bizantinos por parte dos seus componentes.

- **Mecanismo de gestão de recursos** – É apresentado um mecanismo de gestão de recursos genérico baseado em *locks*.

- **Execução especulativa** – São apresentadas e avaliadas duas formas de especulação no cliente, ambas baseadas em futuros.

1.5 Estrutura do Documento

No Capítulo 1 deste documento foi feita uma introdução ao problema que pretendemos tratar nesta dissertação. Foi apresentada a noção base de falhas bizantinas, tal como a

forma como podemos utilizar a replicação como mecanismo de tolerância a este tipo de falhas. Foi ainda apresentada a noção de especulação e a forma como pode melhorar o desempenho deste tipo de sistemas distribuídos.

No Capítulo 2 são apresentados conceitos fundamentais, bem como outros trabalhos que servem de base para esta dissertação. É feito um estudo sobre a forma como o mecanismo de invocações remotas da linguagem Java funciona e trabalho relacionado mais relevante sobre replicação de objectos remotos. É também feito um estudo sobre replicação incluindo as implicações e benefícios desta técnica. Por fim, são apresentadas exemplos de execução especulativa em vários níveis. Ao nível das *threads*, do sistema operativo ou no âmbito dos sistemas de tolerância a falhas bizantinas.

No Capítulo 3 é apresentada a arquitectura da nossa solução. É apresentada o desenho geral de um sistema de tolerância a falhas bizantinas que utiliza uma biblioteca BFT para efectuar as comunicações e utiliza um mecanismo de especulação. É ainda apresentado o JNFS, aplicação utilizada para testar o nosso trabalho. Por último, é apresentado ainda um sistema de contenção que tem como objectivo controlar o acesso aos recursos acedidos pelos servidores.

No Capítulo 4 são apresentados os detalhes da implementação do trabalho desta dissertação. Começamos por apresentar as alterações que é necessário fazer no RMI para mudar o modelo de comunicação. É também apresentada a forma como implementámos uma camada de comunicação baseada em UDP como versão preliminar. Em seguida, é descrita a biblioteca BFT utilizada neste trabalho, tal como a forma como esta é integrada na camada de transporte do RMI, por forma a fornecer tolerância a falhas bizantinas.

No Capítulo 5 são apresentados os testes de validação ao nosso trabalho. É feita uma descrição do *benchmark* utilizado para avaliar o impacto que a tolerância a falhas bizantinas tem nas aplicações RMI. São ainda apresentadas as duas formas de especulação no cliente implementadas: Especulação entre Operações Independentes e Especulação entre Operações Dependentes. Os resultados apresentados revelam a forma como estes mecanismos influenciam a performance dos sistemas em termos de tempo total e de operações efectuadas por segundo.

Por último, no Capítulo 6 são apresentadas as nossas conclusões, tal como algum trabalho futuro.

Trabalho Relacionado

Neste capítulo iremos apresentar algum trabalho relacionado que nos irá ajudar a definir vários aspectos acerca das técnicas e tecnologias utilizadas nesta dissertação.

Em primeiro lugar, na secção 2.1 iremos apresentar o mecanismo de invocações remotas da linguagem Java, o RMI, tal como algumas alternativas à sua implementação. Na secção 2.2 iremos abordar alguns detalhes sobre replicação, no âmbito dos sistemas distribuídos e , em particular, em sistemas tolerantes a falhas bizantinas. Por último, na secção 2.3 iremos apresentar a técnica de execução especulativa, abordando os seus benefícios e implicações.

2.1 RMI

Na linguagem Java existe um mecanismod e invocações remotas chamado RMI (Remote Method Invocation). Este pode ser visto como um modelo de objectos distribuído que visa transportar para as aplicações distribuídas o modelo de programação das aplicações não distribuídas.

A motivação deste mecanismo é precisamente tornar mais simples a construção de aplicações distribuídas, procurando que o programador se abstraia dos detalhes como os dados são transportados entre objectos e do facto de os objectos que compõem a aplicação se encontrarem muitas vezes em máquinas virtuais diferentes e, possivelmente, em máquinas, ou *hosts*, diferentes.

As aplicações distribuídas que fazem uso de RMI têm geralmente uma estrutura composta por um cliente e um servidor. Este último pode ser visto como um objecto remoto que disponibiliza um conjunto de operações através de métodos que podem ser invocados pelos clientes. Estes, antes de conseguirem fazê-lo, precisam de obter uma

referência remota para o servidor, geralmente através do serviço de *naming* do RMI, o `rmiregistry`.

Existem alguns termos utilizados que são importantes definir, de modo a que se compreenda melhor o modelo distribuído do Java:

- **Objecto Remoto** – Um objecto remoto é um objecto onde podem ser efectuadas invocações a partir de outras máquinas. Geralmente desempenha o papel de servidor.

- **Interface Remota** – Uma interface remota é uma interface Java que define os métodos de um objecto remoto.

- **Invocações Remotas** – Chamamos invocações remotas ao acto de efectuar invocações de métodos em objectos remotos. Estas invocações têm exactamente a mesma sintaxe das invocações entre objectos não remotos.

A principal vantagem do RMI é que, para o programador, efectuar invocações de métodos remotos é idêntico a efectuar invocações de métodos num objecto local à sua máquina virtual. A forma como o objecto é encontrado e os dados são enviados é inteiramente responsabilidade do próprio mecanismo, num serviço que é transparente para o utilizador.

Ainda assim, existem algumas diferenças entre o modelo distribuído e o modelo normal de programação do Java. Por exemplo, os argumentos que não representem objectos remotos são todos passados por cópia, uma vez que uma referência se torna inútil numa máquina virtual diferente. É importante também ter a noção de que um cliente interage com uma interface remota, e não com um objecto remoto, de forma a que a implementação do servidor lhe seja ocultada. Outro ponto relevante tem a ver com o facto de o modelo de falhas deste tipo de aplicações ser bastante mais complexo do que o modelo de falhas das aplicações normais, pelo que as aplicações distribuídas são obrigadas a lidar com outro tipo de excepções.

2.1.1 Arquitectura

O RMI pode ser visto como um *middleware* cujo objectivo é esconder das aplicações o facto de os objectos que as compõem se encontrarem em máquinas virtuais diferentes, libertando o programador de todo o trabalho e dos pormenores das comunicações entre objectos. A sua arquitectura é baseada em três grandes camadas:

- **Stub/Skeleton** – Esta camada representa a interface entre as aplicações e o RMI. Os clientes efectuam invocações remotas num objecto através do *stub*, que funciona como uma *proxy* para o objecto remoto e implementa as mesmas interfaces que este. Do lado do servidor, o *skeleton* envia os pedidos recebidos pela camada de referências remotas para os métodos respectivos do objecto remoto. Esta camada é

ainda responsável pelo processo de *marshaling* e *unmarshaling* dos parâmetros das invocações e dos valores de retorno.

- **Referências Remotas** – Esta camada é responsável por resolver os objectos remotos para os quais as invocações são efectuadas e por validar a semântica das mensagens recebidas.

- **Transporte** – Esta camada é responsável pelo transporte das mensagens. Tem como funções estabelecer e manter as conexões entre cliente e servidor. Na implementação de raiz do RMI, estas comunicações são efectuadas através de conexões TCP/IP.

Na Figura 2.1 vemos a forma como estas camadas comunicam entre si. O cliente efectua uma invocação através do *stub*, que consiste numa mensagem que é passada entre as várias camadas até ser entregue na máquina virtual do servidor e, por fim, ao próprio servidor que irá efectuar a operação. No fim desta operação é retornando o seu resultado ao cliente, efectuando o caminho inverso, novamente através das camadas.

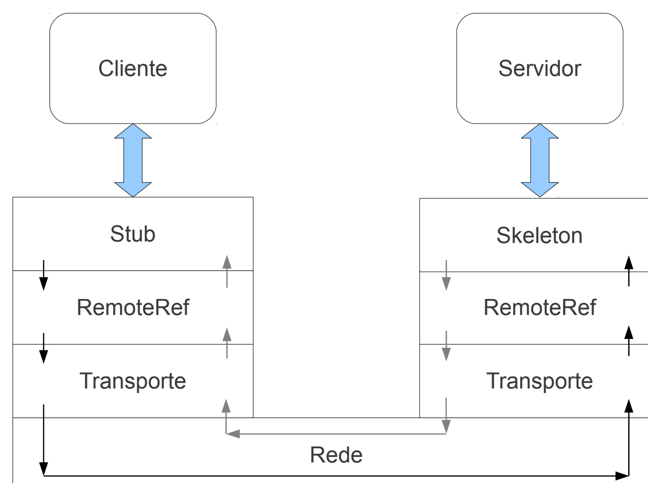

Figura 2.1: Arquitectura do RMI.

2.1.2 Outras Propostas

O modelo do RMI, por se basear numa arquitectura cliente/servidor, tem as suas limitações em termos de fiabilidade, disponibilidade e tolerância a falhas, requisitos bastante importantes num sistema distribuído. Para alcançar estes requisitos é necessário que as aplicações sejam construídas com este objectivo em mente, sendo que toda a arquitectura da aplicação será bastante diferente das aplicações normais cliente/servidor.

A replicação, de dados e de serviços, é uma forma de garantir uma boa resistência a falhas em aplicações distribuídas, criando assim sistemas robustos. Introduzir replicação de objectos remotos em aplicações distribuídas baseadas em RMI pode ser feito de duas formas: (1) criando explicitamente toda a estrutura da aplicação, isto é, criando explicitamente todos os objectos que representam o sistema, tendo em conta todos os detalhes de

comunicação e sincronização entre servidores através da definição de um protocolo de comunicação entre eles; (2) de forma implícita para o programador, sendo toda a replicação de objectos uma responsabilidade interna do RMI. Neste último caso, estaríamos a tirar partido do facto de o RMI ser um *middleware*, uma vez que para o cliente esta estrutura seria invisível e o seu comportamento manter-se-ia, como se estivesse a funcionar sobre uma arquitectura com um único servidor.

O Aroma [NMMS00] é um *middleware* que tem como objectivo melhorar o modelo distribuído do RMI de forma transparente, introduzindo um mecanismo implícito de replicação de objectos remotos. O sistema é composto por várias réplicas distribuídas em máquinas virtuais, e possivelmente em *hosts*, diferentes. É alterada a forma de comunicação entre cliente e servidor, uma vez que, ao ter vários servidores, passamos a ter um modelo de comunicação de um para muitos, em vez do modelo um para um, inerente à arquitectura cliente/servidor. Assim, todas as invocações efectuadas pelo cliente são interceptadas por um componente chamado *Interceptor*, como é visível na Figura 2.2, e que se encontra integrado na camada de transporte do RMI. Os pedidos são depois ordenados de forma total e enviados para todas as réplicas, que irão assim executar os pedidos exibindo o comportamento de uma máquina de estados replicada. Este sistema permite melhorar os mecanismos de tolerância a falhas do RMI mas, no entanto, não apresenta qualquer forma de tolerância a falhas bizantinas.

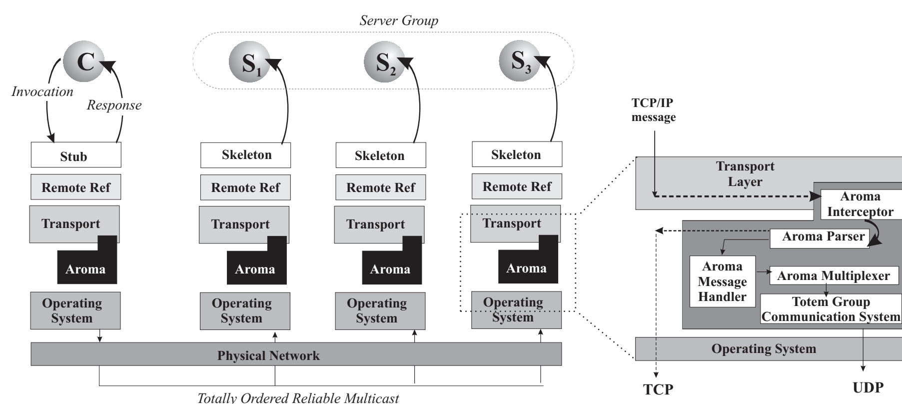

Figura 2.2: Arquitectura do Aroma.(Retirado de [NMMS00])

2.2 Replicação

Em sistemas distribuídos existem requisitos fundamentais para que um sistema tenha um bom comportamento, como uma forte tolerância a falhas e boas garantias de alta disponibilidade. Uma forma de atingir estes objectivos é através da replicação tanto de dados como de serviços.

O facto de um sistema apresentar uma alta disponibilidade é importante, uma vez que os utilizadores cada vez mais exigem ser capazes de aceder a aplicações remotas a qualquer hora e em qualquer lugar. Se os dados estiverem replicados em vários servidores e

estes falharem por causas independentes, no caso de um deles se encontrar indisponível num determinado momento, um cliente pode aceder aos dados através de outro servidor. Além disso, o facto de os dispositivos móveis serem cada vez mais comuns, permite que os utilizadores se movimentem entre redes sem fios, estando várias vezes desconectados. Nestas alturas, os utilizadores podem aceder aos dados através de uma cópia local, o que pode ser visto como uma forma de melhorar a performance, uma vez que os dados são acedidos mais rapidamente. No entanto, é necessário garantir que esta cópia local não se encontra desactualizada, o que implica que periodicamente sejam feitas consultas ao servidor o que, por sua vez, representa um custo acrescido em termos de desempenho.

O facto de um sistema ser tolerante a falhas garante que não só os dados se encontram disponíveis, mas que também se encontram correctos. Por outro lado, sistemas que não tolerem falhas podem garantir a disponibilidade dos dados, mas não a sua correcção. Esta é uma característica que pode ser alcançada através da replicação, o que implica que os vários servidores se coordenem entre eles de forma a garantir a correcção dos dados, especialmente no caso de se considerar a possibilidade da existência de falhas arbitrárias.

2.2.1 Características

A replicação pode ser utilizada em vários tipos de sistemas, com diversas funcionalidades. Por exemplo, os browsers web guardam cópias locais das páginas visualizadas. Ao fazer isto, o browser consegue três coisas. Primeiro, está a melhorar a performance, uma vez que os dados são acedidos mais rapidamente caso se encontrem armazenados localmente. Em segundo, está a garantir a disponibilidade do sistema no caso de o utilizador se desconectar. Por último, garante que no caso de o servidor falhar, o utilizador conseguirá aceder aos dados através da sua cópia local.

Em Sistemas de Gestão de Bases de Dados (SGBDs), os dados são sempre o recurso mais importante e valioso. Consideremos um sistema que requer uma alta disponibilidade e uma forte resistência a falhas, como uma loja de larga escala, onde a segurança dos dados é um ponto crítico. Se houver algum tipo de falha que leve a uma perda dos dados relativos aos clientes ou aos produtos, como encomendas, a loja pode incorrer em perdas graves. Por outro lado, se os clientes não conseguirem contactar a loja, por este se encontrar fora de serviço, podem haver danos graves para a loja em termos de negócio. A replicação surge como uma boa solução para ambos os problemas, uma vez que os dados são replicados, mesmo que através de sistemas RAID, garantindo que não se perdem, e os clientes podem aceder aos serviços através de mais do que um servidor, para o caso de algum falhar.

Podemos também colocar o exemplo de uma companhia cujo objectivo é fornecer um serviço de e-mail a uma escala global. Neste caso os dados não têm um valor tão elevado e a disponibilidade não é um requisito tão forte. No entanto, o facto de os dados se encontrarem replicados por vários pontos no globo permite melhorar a performance e distribuir a carga geral do sistema, uma vez que a companhia pode indicar aos clientes de

uma determinada zona que acedam às suas contas através do servidor que se encontrar mais perto.

Uma das principais motivações da replicação é fornecer tolerância a falhas. Diz-se que um sistema tolerante a falhas é aquele que garante tanto disponibilidade como a correcção dos dados e, muitas vezes, a replicação é a forma mais sensata para atingir ambas. Se considerarmos um modelo de falhas *fail-stop*, isto é, em que se considera que um servidor falha deixando de funcionar e de responder a pedidos, necessitamos de $f + 1$ servidores para garantir os serviços, assumindo que até f servidores podem falhar simultaneamente. No caso mais complexo de assumirmos a presença de comportamentos arbitrários por parte dos servidores, são necessários $3f + 1$ servidores para garantir o serviço, se até f servidores apresentarem um comportamento bizantino. Mais à frente neste capítulo iremos abordar esta questão com maior detalhe.

Apesar dos claros benefícios que podemos obter utilizando replicação, esta pode tornar um sistema distribuído bastante complexo. Se um sistema for composto por várias cópias dos dados e uma delas for modificada, esta modificação terá de se reflectir em todas as outras réplicas, ou acabaremos por ter dados inconsistentes. Por exemplo, retornando ao exemplo da loja, se um utilizador efectuar alguma encomenda, esta terá de ser efectuada em todas as cópias. Caso contrário os armazéns podem apresentar quantidades diferentes de alguns produtos o que não é, de modo algum, o comportamento pretendido embora possa ser tolerado até um certo ponto. Assim, para evitar este tipo de problemas e garantir a correcção dos dados, é necessário que existam protocolos de coordenação, que garantam que as cópias dos dados não diferem umas das outras.

O principal objectivo de um sistema distribuído é fornecer um determinado serviço a um cliente. Este serviço pode ser fornecido por um ou mais servidores, existindo diferenças claras entre as duas aproximações, sendo a mais óbvia a tolerância a falhas. Um sistema que utilize apenas um servidor tem a arquitectura mais simples possível, constituída por um cliente e um servidor. No caso de ser composto por vários servidores, a arquitectura de um sistema pode seguir um de dois modelos de replicação [Sch90, CDK05]:

- **Replicação Passiva** – Neste modelo, o sistema é composto por um servidor primário e um conjunto de servidores secundários, conhecidos como *backups*. Os clientes invocam operações no servidor primário, que as executam e guarda os resultados nos secundários. Este tipo de sistemas toleram até f falhas de um total de $f + 1$ servidores, mas não toleram falhas bizantinas.

- **Replicação Activa** – Neste modelo, quando um cliente invoca uma operação, esta é executada imediatamente em todos os servidores. Normalmente, este modelo segue uma filosofia de máquina de estados replicada, onde todas as operações são executadas em todos os servidores pela mesma ordem, o que implica que haja um acordo entre os servidores sobre a ordem em que devem executar as operações. Para atingir este requisito, a maioria dos sistemas utiliza um servidor primário que dissemina os pedidos pelos restantes servidores [CL02, KAD+07, PRHL08].

2.2.2 Coordenação

Lamport [Lam78] definiu um sistema distribuído como sendo composto por um conjunto de processos fisicamente separados, que comunicam entre eles através de troca de mensagens. Estas mensagens podem ser vistas como eventos que podem mudar os dados e, como tal, têm de ser ordenados de modo a garantir que todos os servidores executam cada operação pela mesma ordem.

Alguns sistemas, como aqueles baseados em máquinas de estados replicadas, requerem que todas as operações sejam executadas pela mesma ordem em todas as réplicas, enquanto que outros sistemas têm requisitos que permitem uma política de ordenação de pedidos menos restrita. É assim importante definir esta política, uma vez que a ordenação de eventos implica trocas de mensagens o que, por seu lado, implica um custo acrescido em termos de performance e tempo de resposta do sistema.

- **Ordem Parcial** – A ordenação parcial é uma forma mais fraca de ordenar eventos, uma vez que apenas alguns destes necessitam de ser totalmente ordenados. Se considerarmos três eventos a, b e c e c tem de acontecer depois de a, então podemos ordenar estes eventos de forma a que a aconteça sempre antes de b ou c, como (a, b, c) ou (a, c, b).

- **Ordem Causal** – A ordenação causal implica que se um evento a pode causar outro evento b, então a tem sempre de ser executado antes de b. Se a e b forem operações do mesmo processo, então a apenas poderá causar b se tiver acontecido antes. Se a e b forem operações de diferentes processos, Pa e Pb, então a apenas poderá causar b se representar a emissão de uma mensagem em Pa e b a sua recepção em Pb.

- **Ordem total** – A ordenação total de eventos implica que estes sejam executados pela mesma ordem em todos os processos. Por exemplo, se um processo executar o evento a e depois b, todos os processos têm de executar a antes de b.

2.2.3 Tolerância a falhas bizantinas

A replicação como forma de alcançar tolerância a falhas bizantinas tem-se tornado numa técnica cada vez mais atractiva, principalmente devido a três factores [KAD+07]. Em primeiro lugar, os dados têm-se tornado num recurso cada vez mais valioso e importante, enquanto que o preço cada vez mais baixo dos componentes de hardware tem tornado cada vez mais fácil, e barato, o armazenamento de dados através desta técnica. Em segundo, as evidências práticas da existência de comportamentos arbitrários em sistemas reais, sugerem que esta técnica pode trazer bastantes benefícios. Por último, os melhoramentos no estado da arte deste tipo de técnicas têm tornado cada vez mais prática a sua utilização e diminuído a diferença de custo para outras técnicas mais comuns.

Um componente, como um servidor num sistema distribuído, que exiba um comportamento bizantino, ou arbitrário, pode falhar de várias formas sem que nunca pare

de executar, continuando sempre a produzir resultados incorrectos, pelo que detectar que uma réplica exibe um comportamento bizantino é uma tarefa complexa. Lamport et al. [Lam82] definiram que, de modo a detectar que f componentes exibem comportamentos arbitrários, é necessário que $3f + 1$ componentes cheguem a essa conclusão através de um consenso. Para isso, as réplicas têm de trocar um número elevado de mensagens, o que leva a que este seja um processo demorado.

Os algoritmos de tolerância a falhas bizantinas devem oferecer garantias de *safety* e *liveness*, de modo a serem considerados correctos.

- **Liveness** – Um sistema que tenha este tipo de garantias assume que irá sempre ser capaz de produzir uma resposta com um resultado correcto a um pedido.

- **Safety** – De forma a garantir segurança, um sistema nunca deve produzir um resultado errado e mostrá-lo ao cliente.

Castro et al. [CL02], propuseram um algoritmo prático (PBFT - *Practical Byzantine Fault-Tolerance*) que oferece garantias de *safety* e *liveness* se até $\lfloor \frac{n-1}{3} \rfloor$ réplicas, de um total de n, falharem em simultâneo. O PBFT é visto como uma máquina de estados replicada e pode ser utilizado para implementar qualquer serviço replicado, desde que seja determinístico, como bases de dados distribuídas ou sistemas de ficheiros distribuídos.

As réplicas avançam através de uma sucessão de *views*, em que em cada uma delas existe uma réplica primária e várias secundárias. O algoritmo tem início no momento em que um cliente invoca um pedido na réplica primária, que dá início a um protocolo de três fases em conjunto com as réplicas, conhecidas como *pre-prepare*, *prepare* e *commit*. Na primeira fase, a réplica primária cria um número de sequência, único dentro de uma *view*, para o pedido que é enviado para as restantes réplicas. Nesta altura, as réplicas dão início à fase de *prepare* enviando uma mensagem para todas as outras indicando que estão preparadas para executar o pedido. Dizemos que uma réplica se encontra *prepared* apenas quando tiver recebido $2f$ mensagens de *prepare*, em que f é o número máximo de réplicas bizantinas no sistema. Estas duas fases servem para garantir que os pedidos se encontram totalmente ordenados dentro da mesma *view*, mesmo que a réplica primária se encontre a falhar. Por último, as réplicas trocam mensagens de *commit* e, após receberem $2f + 1$ destas mensagens, executam o pedido e devolvem o resultado final da operação ao cliente, que apenas considera o resultado válido no momento em que receber $f + 1$ respostas consistentes, isto é, com o mesmo resultado. Como podemos observar na Figura 2.3, este protocolo leva a uma elevada troca de mensagens, o que prejudica a performance do sistema.

De um modo geral, as falhas neste tipo de sistemas são raras [WCN+09] e, como tal, a resposta de uma única réplica pode representar uma boa previsão do que vai ser o resultado final de uma operação. No caso de uma operação ter um resultado bastante previsível, os próprios clientes podem especular num resultado final, antes de receberem qualquer resposta de qualquer réplica. Na secção 2.3.3 iremos aprofundar este assunto e

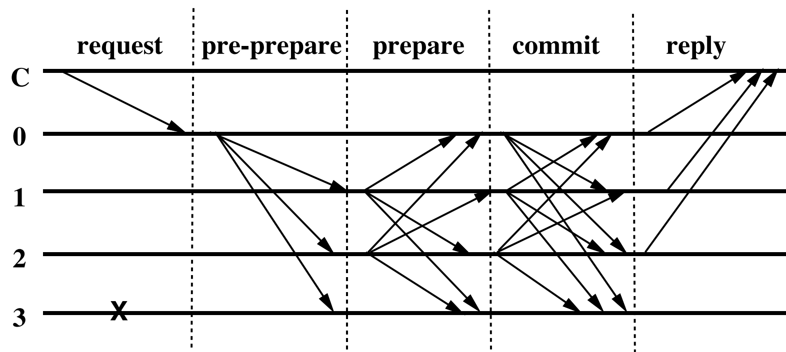

Figura 2.3: Execução normal do algoritmo PBFT. (Extraído de [CL02])

mostrar exemplos de como a especulação pode ser uma técnica benéfica para este tipo de sistemas.

2.2.4 Byzantium

O Byzantium [PRHL08] é um *middleware* de replicação de bases de dados com tolerância a falhas bizantinas que permite que as operações sejam executadas concorrentemente, melhorando a performance, e que não contém nenhum componente centralizado, garantindo a segurança do sistema e a sua disponibilidade. Sendo constituído por um *middleware*, o Byzantium permite a utilização de sistemas de bases de dados já existentes sem qualquer modificação. Permite ainda a utilização de diferentes implementações em diferentes réplicas, o que aumenta a resistência do sistema contra falhas derivadas de vulnerabilidades presentes nos programas.

As bases de dados distribuídas comuns têm uma arquitectura distinta do Byzantium. Geralmente o controlo do sistema é responsabilidade de um único sistema de gestão de bases de dados e os dados encontram-se replicados em diferentes localizações. O principal objectivo destes sistemas é garantir que os dados se encontram duplicados, garantindo a sua protecção em caso de falhas. Outro dos objectivos é melhorar a performance, colocando a informação mais perto de quem necessita. Por exemplo, uma empresa pode colocar fragmentos da base de dados espalhados pelas várias secções, facilitando o acesso e controlo dos dados.

Ainda que um dos principais objectivos das bases de dados distribuídas seja melhorar a disponibilidade e fiabilidade, normalmente estas não possuem mecanismos de tolerância a falhas bizantinas. Estes sistemas também são vulneráveis a falhas relacionadas com o software, como falhas de segurança ou *bugs*. Por exemplo, se houver um problema de segurança com uma das bases de dados, todo o sistema ficará vulnerável. O Byzantium fornece maiores garantias de segurança, uma vez que permite que diferentes implementações de sistemas de gestão de bases de dados sejam utilizadas em diferentes réplicas, garantindo a independência das falhas específicas dessas implementações.

De forma a aumentar a concorrência entre operações, o Byzantium apenas fornece

garantias de *Snapshot Isolation*. Desta forma, cada transacção executa contida numa visão, ou *snapshot*, da base de dados e pode terminar com sucesso, efectuando *commit*, se não provocar conflitos de escrita/escrita com outra transacção concorrente. Sendo apenas necessário evitar conflitos de escrita/escrita, transacções que apresentem conflitos de escrita/leitura ou leitura/leitura podem ser executadas concorrentemente, o que permite aumentar a performance global do sistema.

A arquitectura do sistema, como podemos ver na Figura 2.4, é composta por um número finito de clientes e $3f + 1$ réplicas e cada operação é vista como uma transacção composta pela operação de `COMMIT`, seguida de vários pedidos de leitura ou escrita, terminando com uma operação de `COMMIT` ou `ROLLBACK`.

Um dos pontos importantes deste género de sistemas é o facto de os pedidos terem de ser ordenados de forma total. Para atingir este requisito, o Byzantium utiliza o protocolo PBFT, embora que com algumas diferenças [PRHL08]. De modo a evitar que os pedidos tenham de passar sempre por este processo, existem duas formas de o cliente contactar o servidor. Para operações que tenham de ser executadas dentro da mesma *snapshot* da base de dados, como `BEGIN` ou `COMMIT`, é utilizado uma biblioteca BFT, como é possível ver na figura 2.4, que irá dar inicio ao protocolo de ordenação. Por outro lado, operações que não necessitem de ser ordenadas, como `reads` ou `writes`, por estarem contidas dentro de uma transacção, o pedido é enviado para todas as réplicas e executado sem que seja preciso passar pelo algoritmo de ordenação, sendo o seu resultado validado mais tarde na operação de `COMMIT`.

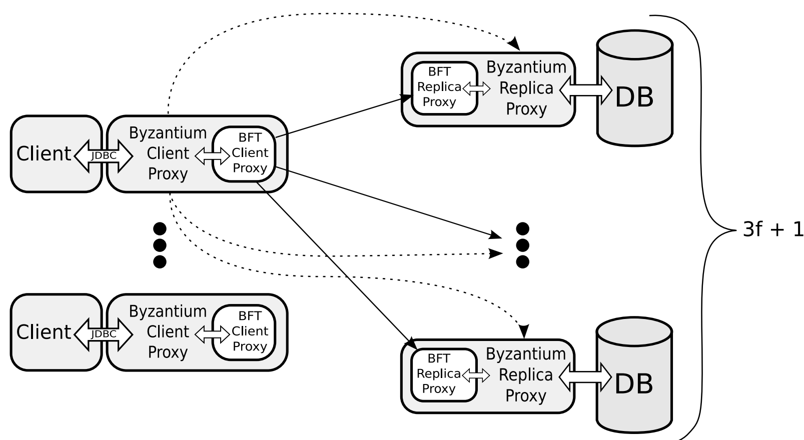

Figura 2.4: Arquitectura do Byzantium.

Na ausência de falhas, o protocolo inicia-se quando um cliente efectua um pedido de `BEGIN` para todas as réplicas. Uma das réplicas será considerada como coordenadora da transacção. Cada transacção é iniciada em todas as réplicas dentro da mesma *snapshot* da base de dados e, nesta altura, podem ser efectuadas operações de leitura e de escrita. Uma operação de escrita será enviada para todas as réplicas, mas será apenas executada pela réplica coordenadora, que devolve o resultado ao cliente. Por outro lado, uma operação de leitura pode ser executada de duas formas diferentes. No caso de se tratar de

uma transacção composta por leituras e escritas, as operações de leitura são executadas da mesma forma que as de escrita. Caso contrário, em transacções *read-only*, uma operação de leitura é executada em $f] + 1$ réplicas escolhidas aleatoriamente e a primeira resposta será enviada para o cliente. Este resultado será mais tarde validado na altura da operação de `COMMIT` e será considerado correcto se as $f + 1$ respostas forem consistentes. Esta optimização representa uma forma de balanceamento da carga, uma vez que as operações apenas executam em $f + 1$ das $3f + 1$ réplicas. Para tirar o máximo partido desta optimização o Byzantium classifica todas as transacções como *read-only* até que seja encontrada uma operação de leitura.

Uma transacção termina apenas quando um cliente efectua um pedido de `COMMIT`, que inclui os sumários, ou *digests*, de todas as operações efectuadas e os seus resultados. Nesta altura, todas as réplicas, excepto a primária, executam as operações e confirmam que os resultados obtidos estão de acordo com aqueles obtidos pela réplica coordenadora, garantindo ainda que todas as transacções podem ser finalizadas, segundo *Snapshot Isolation*.

2.3 Especulação

Muitas vezes num programa sequencial, existem algumas computações longas que acabam por atrasar todo o programa. Apesar de não poderem ser evitadas, muitas destas computações têm resultados bastante previsíveis e, nestas situações, podemos utilizar execução especulativa como forma de melhorar a performance.

A execução especulativa tem como objectivo evitar longos períodos de espera pelo resultado de uma operação. No caso de termos de executar uma operação longa com um resultado previsível, podemos especular no seu valor final, ao invés de esperar pelo seu fim. Este valor será considerado um valor temporário, uma vez que terá de ser validado mais tarde, quando a operação terminar.

Quando especulamos acerca do resultado final de uma operação de forma errada e o valor temporário é utilizado por outras operações, estas terão de ser executadas novamente com base no valor correcto, de forma a evitar inconsistências. Por isso, num mecanismo de especulação, é importante que exista alguma forma de retornar à instrução onde se iniciou a especulação, anulando as operações efectuadas com base no valor incorrecto.

A principal vantagem do uso de execução especulativa é o facto de podermos antecipar a execução de código que sucede uma operação muito demorada. Por exemplo, num sistema distribuído, quando um cliente efectua um pedido a um servidor terá de esperar que o pedido chegue ao mesmo, onde é calculado o seu resultado que será enviado de volta ao cliente. Estes passos podem levar muito tempo e, no caso de ser uma operação com um resultado previsível, como um `insert` numa base de dados, o cliente pode especular o seu valor final e continuar a executar as operações seguintes, evitando um longo tempo de espera.

As arquitecturas *multi-core* e os computadores com vastos recursos existentes hoje em dia, facilitam o uso de especulação. Com processadores *multi-core*, por exemplo, podemos executar uma operação a executar num dos núcleos, enquanto que noutro núcleo executamos outras operações baseadas num valor especulativo e, dado que muitas vezes não fazemos uso de todos os núcleos, estaremos a aproveitar um recurso que, de outra forma, seria desperdiçado.

Apesar do seu grande potencial, a execução especulativa pode trazer alguns problemas, se não for utilizada de forma cuidada. Nightingale et al. [NCF06] enumera três condições fundamentais para a utilização de especulação:

- **Os resultados das operações têm de ser bastante previsíveis** – Quando uma previsão está errada, todas as operações que se basearam no seu valor terão de ser novamente executadas, o que poderá resultar num desperdício de recursos. Assim, é importante que sejam feitas previsões acertadas, de modo a reduzir ao mínimo o número de operações executadas mais do que uma vez.

- **A operação de *checkpointing*** – No caso de termos uma especulação falhada, o programa deve retornar ao estado anterior à operação e anular as alterações feitas durante a execução especulativa. Desta forma, é importante que a operação de guardar o estado do programa, ou *checkpointing* não seja mais demorado do que a operação em si.

- **Existem vários recursos livres nos computadores** – A execução especulativa necessita de ciclos do processador, enquanto que a operação de *checkpointing* requer memória. Hoje em dia estes recursos existem em abundância, sendo até muitas vezes desperdiçados, facilitando o uso desta técnica.

2.3.1 Especulação ao nível das Threads

Beneficiando das arquitecturas *multi-core*, a especulação ao nível das *threads* é uma das formas mais comuns de especulação. Ao longo deste capítulo iremos referir-nos a esta forma de especulação como TLS (Thread Level Speculation). Nesta técnica, as *threads* são utilizadas para encapsular diferentes fluzos de execução especulativa, transformando um programa habitualmente executado de forma sequencial, isto é, com um único fluxo de execução, num programa com vários fluxos especulativos, com o objectivo de melhorar a sua performance. Muitas vezes esta não é uma tarefa simples, principalmente devido a dois factores. Em primeiro lugar, prever o resultado final de uma operação pode não ser simples e pode gerar desperdício de tempo. Em segundo, temos de ter em consideração as dependências entre *threads*, uma vez que estas podem partilhar e modificar dados.

Oplinger e Lam [OL02] usaram TLS para melhorar programas de monitorização de software. Estes programas, que muitas vezes são fulcrais para garantir fiabilidade em software crítico, acrescentam um custo significativo em termos de performance e, se não forem encontrados quaisquer erros, o uso de funções de monitorização dos programas

leva apenas a um desperdício de tempo. Assim, uma vez que estas funções não têm qualquer efeito no estado do programa, a não ser que sejam detectadas anomalias, podem ser executadas especulativamente, assumindo que não existem erros, tirando partido de vários processadores que estejam disponíveis. O uso de TLS torna assim possível que existam múltiplos fluxos de execução de cada vez, comportando-se cada um como uma transacção, abortando quando existirem erros.

Locasto et al. [LK06] também abordaram o mesmo problema, embora utilizando uma aproximação distinta. Através de uma camada de virtualização, podem seleccionar fragmentos de código para ser executado. Esta camada é composta por um conjunto de *virtual executors*, que são responsáveis pela execução das operações. Desta forma, um fragmento de um processo, como uma função de monitorização, pode ser seleccionada para executar de forma especulativa em concorrência com o programa principal.

Chang et al. [CG99] propuseram uma forma de alterar aplicações que usam o disco de forma intensiva de modo a que executem especulativamente (Figura 2.5). Enquanto esperam pelo resultado de uma operação de *I/O*, um programa pode executar previamente as operações seguintes e, sempre que uma delas for uma leitura, tentar colocar os dados em memória, onde podem ser acedidos mais rapidamente, assumindo que vão realmente ser necessários mais tarde. No entanto, no caso de especulação estar errada, vão ser lidos dados desnecessários, ocupando os discos durante algum tempo, o que pode prejudicar outras aplicações.

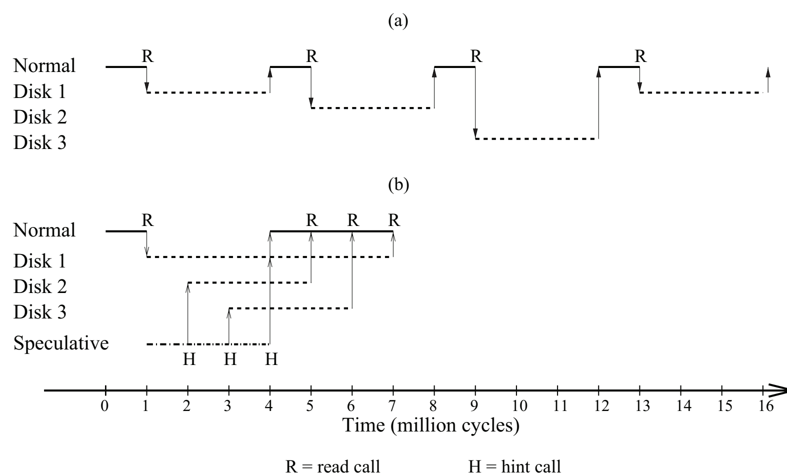

Figura 2.5: Exemplo de execução especulativa. (a) mostra a execução normal, sem especulação. (b) mostra a forma como a especulação pode reduzir o tempo de espera. (Retirado de [CG99])

Steffan et al. [SCZM00] utilizaram TLS para melhorar a performance em algumas aplicações paralelizando excertos de código. Esta proposta tem a vantagem de ser escalável em qualquer máquina, quer seja em processadores com um único chip ou em máquinas de larga escala. Resumidamente, existem excertos de código, como os ciclos, que podem ser paralelizados, executando cada iteração concorrentemente. Obviamente que várias iterações do mesmo ciclo podem partilhar dados, criando dependências entre elas que

podem ser violadas, por exemplo, se uma iteração `j` tenta ler dados que serão mais tarde modificados por uma iteração anterior `i` a executar numa *thread* diferente. Neste caso, existirá um conflito que será resolvido voltando a executar a iteração `j`, utilizando o valor correcto produzido na iteração `i`. De modo a detectar estes problemas em tempo de execução, são usados mecanismos de coerência de cache.

Li et al. [LDZN03] procuraram melhorar o paralelismo ao nível das instruções através de paralelismo entre *threads*. É assumida uma arquitectura com dois processadores em que um é considerado o principal, onde será executada a *thread* principal do programa, e o outro o secundário, ou processador especulativo, onde serão executadas as *threads* especulativas. No momento em que a *thread* principal chega a um ponto em que precisa de alguns dados que ainda não foram calculados, prevê o seu valor e continua a executar especulativamente sem paragens, enquanto o resultado correcto será calculado em paralelo, no processador secundário. De modo a decidir que valores podem ser previstos, o programa passa por um processo de compilação dividido em dois passos, em que o compilador tenta minimizar o custo da especulação seleccionando apenas instruções cujo resultado seja previsível.

A especulação ao nível das *threads* pode também ser utilizada no desenho de processadores. Marcuello e González [MG99] apresentaram uma micro-arquitectura com a capacidade de executar múltiplos fluxos em simultâneo. É composta por várias unidades de processamento que partilham apenas alguns registos, sendo a maioria dos recursos local a cada unidade. Este desenho facilita a localidade das comunicações, o que reduz os seus custos, e não necessita de qualquer suporte por parte do compilador, uma vez que a previsão de valores e as verificações de dependências entre *threads* são efectuadas pelo hardware. Desta forma, o processador pode extrair múltiplas *threads* especulativas de um programa sequencial, composto por um único fluxo de execução, e colocá-las a executar uma unidade de processamento disponível.

Seguindo o crescente interesse nas arquitecturas GPU, Liu et al. [LEG10] propuseram um modelo de execução especulativa e previsão de valores para GPUs. De modo a amortizar os custos de comunicação e sincronização, a especulação em processadores requer *threads* com uma granularidade mais alargada o que, no caso de tratar de uma granularidade demasiado larga, necessita de demasiado armazenamento para os estados especulativos. Os GPUs fornecem *closely coupled threads* e vários registos para cada uma delas, o que reduz de forma significativa o custo das comunicações e da sincronização entre *threads*.

2.3.2 Especulação ao nível do Sistema

Nightingale et al. [NCF06] criaram o Speculator com o objectivo de tornar os sistemas de ficheiros distribuídos mais eficientes através do uso de execução especulativa, fornecendo suporte para esta técnica no kernel do Linux. Neste tipo de sistemas, como na maioria dos sistemas distribuídos, as comunicações representam um elevado custo que pode

ser reduzido executando especulativamente algumas operações cujo resultado é bastante previsível.

De forma a reduzir o custo das comunicações, muitas implementações de sistemas de ficheiros distribuídos apenas fornecem garantias de consistência fracas uma vez que as verificações de coerência dos dados implicam comunicações síncronas entre os vários componentes do sistema. O uso de execução especulativa permite não só reduzir o impacto destas comunicações, como aumentar as garantias oferecidas pelo sistema. Por exemplo, quando um cliente invoca uma operação num servidor, não recorrendo a especulação, irá bloquear-se enquanto espera por uma resposta mas, se especularmos sobre o resultado da operação, o cliente poderá continuar a executar as operações seguintes, poupando o tempo de retorno do resultado e passando a comportar-se de forma assíncrona. Podemos ver este exemplo na figura 2.6.

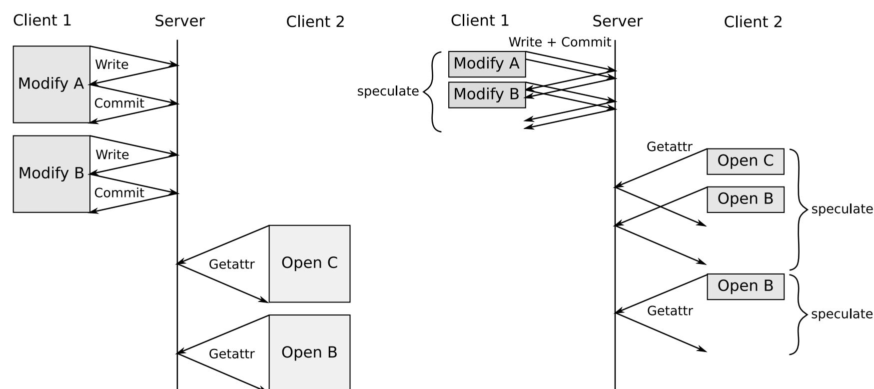

Figura 2.6: Speculator. (Retirado de [NCF06].)

Cully et al. [CLM+08] utilizaram especulação com o objectivo de melhorar a performance dos mecanismos de *checkpointing* presentes em máquinas virtuais, criando o Remus. Este tipo de mecanismos fornecem elevados graus de disponibilidade e tolerância a falhas. No entanto, implicam que o estado da máquina virtual seja replicado na sua totalidade para uma máquina remota, o que pode ser um processo demorado. Em vez de esperar pelo fim deste processo, o Remos escreve os dados alterados para um buffer e continua a execução de forma especulativa, sem que sejam externalizados quaisquer dados especulativos.

Ambos os trabalhos apresentados, embora que em âmbitos distintos, fizeram uso de execução especulativa como forma de melhorar a performance nos respectivos sistemas. Em particular, em [NCF06] foi possível melhorar substancialmente o desempenho dos sistemas de ficheiros testados o que se deve, em grande parte, à alteração dos padrões de comunicação destes sistemas para um modelo assíncrono.

2.3.3 Especulação em sistemas de tolerância a falhas bizantinas

Nos sistemas de tolerância a falhas bizantinas (BFT), o custo das comunicações entre réplicas é bastante elevado. Nestes sistemas, de forma a evitar comportamentos arbitrários,

e possivelmente erróneos, por parte das réplicas, é necessário que estas cheguem a um consenso relativamente aos resultados das operações. Este processo é geralmente bastante demorado, o que leva a um aumento acentuado da latência do sistema e, como tal, o uso de especulação pode ajudar a diminuir o seu impacto na performance global do sistema.

A forma mais comum de especulação dá-se no cliente, isto é, a responsabilidade de manter estados especulativos e de validar a sua correcção é inteiramente do cliente. Liskov et al. [WCN+09] apontaram duas formas de especular no cliente:

- **Baseado em 1 resposta** – O cliente especula com base na primeira resposta a um pedido, que recebe por parte de uma réplica.
- **Baseado em 0 respostas** – O cliente faz uma previsão do resultado final de uma operação, ainda antes de receber qualquer resposta. É importante frisar que este tipo de especulação apenas pode ser utilizado em operações altamente previsíveis.

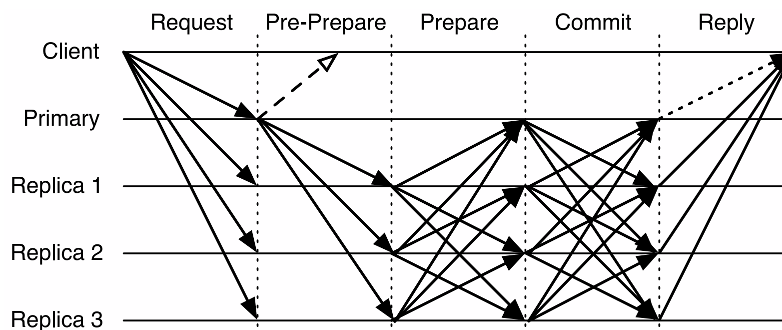

Figura 2.7: Execução normal do algoritmo PBFT-CS. (Extraído de [WCN+09])

Na Figura 2.7 podemos encontrar a forma como o algoritmo PBFT-CS (*Practical Byzantine Fault-Tolerance - Client Speculation* que usa especulação da parte do cliente. Neste algoritmo, ao contrário da sua versão anterior, no momento em que recebe um pedido, a réplica primária executa-o imediatamente e envia o seu resultado especulativo para o cliente, que será o resultado correcto no caso de a réplica se encontrar a executar sem falhas. Ao receber este valor especulativo, o cliente começa imediatamente a executar as operações seguintes. Uma vez que estas podem depender do resultado anterior, o cliente mantém uma lista de todas as operações cujo resultado é provisório, de modo a que não seja possível finalizar, ou fazer *commit*, de operações baseadas em resultados especulativos.

No caso de uma especulação se encontrar errada, isto é, quando o seu valor não corresponder ao resultado final da operação, o cliente tem de voltar a executar as operações que dependem deste valor. Geralmente, quando esta situação se verifica, é um bom indicador de que a réplica primária se encontra num estado bizantino.

Em [KAD+07] é utilizada uma ideia semelhante. O Zyzzyva iguala o mínimo teórico de $3f+1$ réplicas, no que toca ao número total de réplicas que participam no protocolo,

tal como o número de réplicas que devem manter o estado global da aplicação e executar as operações. É visto como uma máquina de estados replicada baseada em três sub-protocolos: (1) *agreement*, cujo objectivo é ordenar os pedidos para serem executados pelas réplicas; (2) *view change*, que acontece quando o sistema conclui que a réplica primária exibe um comportamento bizantino, levando a que seja eleita outra réplica para desempenhar o seu papel; (3) *checkpoint*, que procura limitar os dados guardados em cada réplica de modo a reduzir o custo das mudanças de *view*.

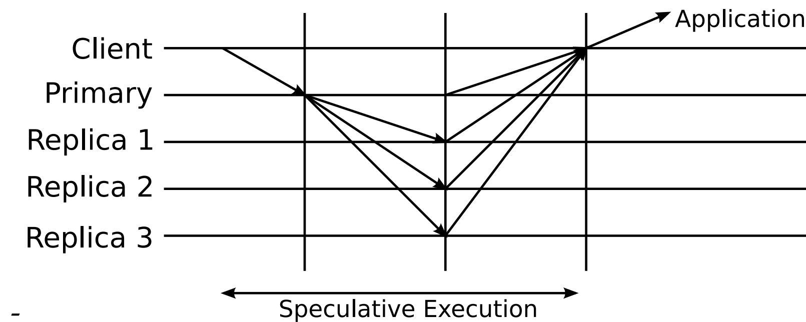

Figura 2.8: Execução normal do algoritmo Zyzzyva. (Extraído de [KAD+07])

O Zyzzyva, tal como o PBFT, está organizado numa sequência de *views*, sendo que em cada uma delas existe uma réplica primária. Ao invés de passar imediatamente pelo algoritmo de ordenação, as réplicas executam primeiro a operação e enviam o seu resultado para o cliente. É, assim, responsabilidade do cliente validar o resultado final de uma operação. Esta validação pode ser feita de duas formas. No caso de receber $3f + 1$ respostas iguais, a operação é considerada completa e o resultado considerado definitivo. No caso de receber entre $2f + 1$ e $3f + 1$ respostas consistentes, o cliente envia para as réplicas um pedido de *commit* e considera o resultado válido quando recebe $2f + 1$ respostas das réplicas a confirmar o valor.

Os trabalhos apresentados nesta Secção utilizam execução especulativa como forma de diminuir o impacto das comunicações em sistemas de tolerância a falhas bizantinas. São implementadas formas distintas de execução especulativa no cliente o que permite melhorar a performance global, diminuindo o tempo que este se encontra à espera dos resultados das operações. Outro ponto relevante para este tipo de sistemas é o facto de as previsões falhadas representarem um indicador fiável da existência de comportamentos bizantinos, especialmente no caso de uma previsão de uma réplica diferir do resultado final, obtido pelas restantes réplicas.

Arquitectura

O mecanismo de invocações remotas do Java, o RMI, permite criar aplicações distribuídas com base no paradigma de programação orientada a objectos. Estas aplicações são desenvolvidas com uma arquitectura cliente/servidor, ou seja, são compostas por um cliente que efectua operações num servidor que disponibiliza um conjunto de operações definido numa interface.

Esta arquitectura transporta para as aplicações todas as suas características. Por um lado, permite-nos alcançar um elevado nível de performance, uma vez que as interacções entre os componentes requerem a troca de um número mínimo de mensagens. Por outro, tendo os serviços e os dados concentrados apenas num servidor, a tolerância a falhas destes sistemas é bastante reduzida, uma vez que este representa um ponto de falha único.

Sendo certo que a performance é um dos principais requisitos de qualquer sistema, não é menos verdade que a segurança dos dados e a disponibilidade dos serviços são características cada vez mais importantes nos sistemas distribuídos actuais. Para lidar com este problema pode recorrer-se a replicação de dados e serviços, em vários servidores. O objectivo do nosso sistema passa por utilizar a replicação de servidores RMI como forma de fornecer tolerância a falhas bizantinas neste tipo de aplicações o que, como já discutimos anteriormente, implica que seja necessário um número relativamente elevado de servidores, ou réplicas, e que estes troquem um elevado número de mensagens para garantir a ordenação total das operações.

3.1 Desenho Geral

Ao utilizar algoritmos de tolerância a falhas bizantinas incorremos numa clara perca de performance e um elevado aumento da latência dos sistemas, sobretudo devido ao elevado número de mensagens trocadas entre as réplicas ainda antes de executarem as operações. Estes sistemas, na sua grande maioria, são compostos por $3f + 1$ réplicas, sendo f o número de réplicas que podem exibir comportamentos bizantinos, e podem ser vistos como uma máquina de estados replicada, o que implica que exista uma ordenação total dos pedidos.

Uma possível forma de diminuir este impacto negativo nos sistemas é a utilização de execução especulativa. Esta técnica, que permite mascarar a latência de um sistema, já foi utilizada por outros sistemas semelhantes [KAD+07, WCN+09]. Estes sistemas utilizam especulação no cliente, permitindo que este efectue operações com base em valores previstos. Estes valores podem ser baseados na execução da operação numa única réplica, assumindo que a ordem em que é executada é a correcta. O principal benefício desta técnica é permitir minimizar o impacto que os algoritmos de ordenação de pedidos trazem para estes sistemas.

Outra forma possível de minimizar os efeitos impostos pela ordenação total dos pedidos é a sua execução concorrente nos servidores. De um modo geral, os pedidos são executados nos servidores de um modo sequencial de forma a que seja mantida a sua ordem. Esta é uma restrição necessária para manter a consistência entre as várias réplicas. No entanto, se tivermos em conta que existem vários pedidos que são independentes, muitos deles poderão ser executados de forma concorrente, melhorando a latência e a performance do sistema, desde que sejam evitados possíveis conflitos.

A arquitectura do nosso sistema é, assim, composta por três componentes principais. O cliente, que pretende executar um conjunto de operações disponibilizadas pelos servidores. Estes têm como principal objectivo executar as operações efectuadas pelo cliente, fornecendo tolerância a falhas bizantinas. Por último, estes dois componentes comunicam através de uma biblioteca BFT que é responsável pela ordenação dos pedidos. Assim, quando o cliente pretende executar uma operação nos servidores, esta é propagada através da biblioteca BFT que se encarrega da sua entrega aos servidores. Estes executam a operação na ordem correcta e retornam o resultado novamente através da biblioteca BFT que, desta forma, também é responsável pela sua entrega ao cliente.

3.2 Estrutura da Solução

A solução proposta nesta dissertação baseia-se na extensão do modelo BFT tradicional com especulação no cliente e execução concorrente no servidor. O modelo proposto foi aplicado à replicação de serviços RMI como forma de alcançar uma solução eficiente de tolerância a falhas bizantinas nestes sistemas.

3.2.1 Especulação baseada em futuros

Como já referimos anteriormente, a execução especulativa é uma técnica que permite melhorar a performance de alguns sistemas, mascarando a sua latência. Como tal, num sistema distribuído, onde muitas vezes grande parte do tempo é despendido em comunicações, esta pode ser uma forma de melhorar a eficiência.

Uma forma possível de utilizar especulação é através de futuros. Um futuro pode ser visto como um objecto que representa uma operação, funcionando como uma *proxy*, enquanto esta se encontra a ser executada concorrentemente com as operações seguintes. Na Figura 3.1 podemos observar a forma como este mecanismo funciona. Quando é necessário executar uma computação, denominada por método alvo, é criado um futuro, enquanto a operação será executada em paralelo com o corpo principal do programa, que se encontra a executar de forma especulativa. Desta forma, será possível avançar com as operações seguintes, independentes da primeira poupando o tempo que estaríamos à espera do seu resultado. Quando for necessário obter o resultado final da operação especulativa, duas situações podem ocorrer: (1) O seu resultado já é conhecido e retornado ao programa principal; (2) A operação ainda não terminou, pelo que temos de esperar pelo seu resultado final, bloqueando o programa principal.

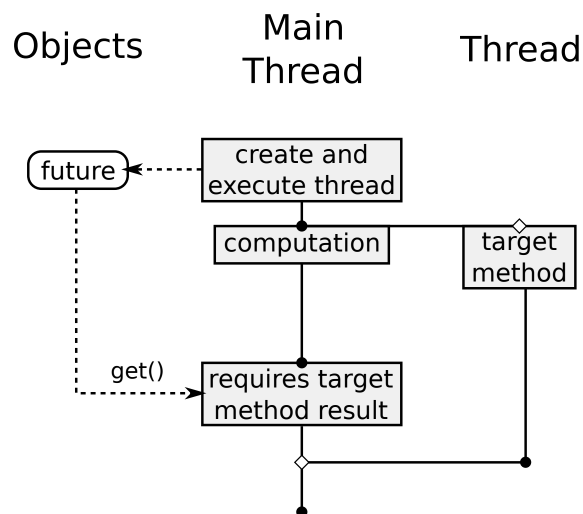

Figura 3.1: Exemplo de especulação baseada em futuros. (Extraído de [Cou])

3.2.2 Replicação com tolerância a falhas bizantinas

Na sua implementação de raiz, as aplicações construídas recorrendo ao RMI seguem uma arquitectura cliente/servidor, sendo que estes se encontram ligados através de uma conexão TCP/IP. No nosso trabalho alterámos esta arquitectura em dois pontos principais. Primeiro, utilizamos servidores replicados para executar os métodos invocados pelo cliente. Segundo, a comunicação entre cliente e servidores passa a ser efectuada através de um algoritmo de tolerância a falhas bizantinas. Em termos práticos, estas alterações

só serão feitas dentro da camada de transporte do RMI, não implicando alterações no cliente ou no servidor.

Ao passarmos a ter os servidores replicados, em vez de termos comunicações de um para um, em *unicast*, as comunicações são feitas entre um cliente e vários servidores, ou seja, através de *multicast*. No nosso trabalho, estas comunicações são feitas através de uma biblioteca BFT que se encontra integrada na camada de transporte do RMI. Na Figura 3.2 vemos como os dados provenientes tanto do cliente como dos servidores passam primeiro pelo RMI, sendo enviados através da biblioteca BFT. Esta biblioteca, como iremos ver com maior detalhe no Capítulo 4, é responsável por ordenar as mensagens antes que os pedidos sejam executados pelos servidores e por mascarar comportamentos bizantinos no sistema

Quando é enviada uma mensagem, como uma invocação de um método por parte de um cliente, esta vai ser encapsulada numa mensagem BFT que é enviada através da biblioteca para todas as réplicas. Antes de ser executado o método, a mensagem terá de passar por um processo de ordenação total em que participam todas as réplicas.

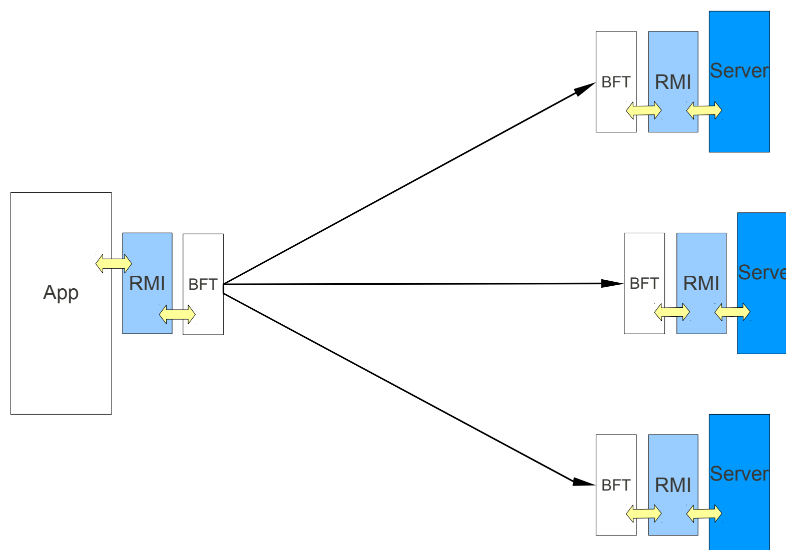

Figura 3.2: Arquitectura de uma aplicação RMI, utilizando a biblioteca BFT.

No final da execução das operações, cada servidor irá enviar a resposta para o cliente através da biblioteca BFT que, ao receber a mensagem que contém o resultado, faz a sua entrega ao RMI que, por sua vez, encaminha o resultado para a aplicação.

3.2.3 Execução concorrente no servidor

A concorrência no acesso aos dados é um dos problemas críticos e mais comuns em muitas aplicações, como sistemas de ficheiros ou bases de dados. Quando utilizamos replicação nestes sistemas torna-se necessário garantir que as réplicas alcançam sempre o mesmo estado o que é complicado caso haja operações concorrentes. Como tal, para que estas aplicações funcionem correctamente, é fulcral que exista alguma forma de garantir

que o acesso aos dados é ordenado e não irá gerar conflitos.

Os algoritmos BFT fazem uma serialização dos pedidos de forma a que estes sejam executados de modo sequencial, eliminando a concorrência entre operações. Esta é uma medida que visa manter a consistência dos dados nas réplicas. No entanto, pode ser algo excessiva se tivermos em conta, por exemplo, operações que não produzem qualquer alteração no estado do sistema, como operações de leitura.

No entanto, a biblioteca BFT utilizada no nosso trabalho permite que as operações sejam executadas concorrentemente, não forçando a serialização dos pedidos. Note-se que os pedidos continuam a ser ordenados de forma total podendo, no entanto, ser executados simultâneamente.

A juntar a este facto, está a forma como o RMI faz o atendimento dos pedidos, criando uma *thread* por invocação. Assim, se tivermos várias invocações remotas em simultâneo, será criada uma *thread* para executar cada uma delas concorrentemente. É, assim, possível que existam conflitos no acesso aos dados por parte das várias *threads*, o que pode levar à divergência entre réplicas, uma vez que não podemos controlar a ordem pela qual as *threads* são escalonadas por parte do sistema operativo. Esta pode assim revelar-se uma fonte de não determinismo na execução das operações, o que é algo que é necessário evitar neste tipo de sistemas.

Assim, é necessário que exista um mecanismo gestão operações por parte da aplicação, que permita evitar conflitos e garantir a ordem das operações. Na Figura 4.15 é apresentada a interface do nosso mecanismo de gestão de acesso a dados, `ContentionManager`.

```
1
2  public interface ContentionManager{
3
4    public static boolean acquireLocks(List<Lock> l);
5
6    public static void releaseLocks(List<Lock> l);
7
8  }
```

Figura 3.3: Interface do mecanismo de gestão de recursos.

Este mecanismo é composto por dois métodos:

- **`acquireLocks(List<Lock> l)`** – Permite que uma *thread* reserve o acesso a um determinado conjunto de recursos.
- **`releaseLocks(List<Lock> l)`** – Faz com que seja libertado o acesso a um conjunto de recursos.

A utilização deste mecanismo leva a que o modelo de programação das operações no servidor sofra algumas alterações. Como é visível no exemplo da Figura 3.4, o conjunto de instruções que representam uma operação deverá ser executado apenas quando for garantido que não haverá conflitos, ou seja, após o retorno do método `acquireLocks`,

que é um método bloqueante. No final das operações deverá ser libertado o recurso, através do método `releaseLocks`, de modo a que outros pedidos concorrentes possam ser executados.

A política de contenção deste mecanismo baseia-se essencialmente em duas premissas. A primeira está relacionada com a ordenação imposta pela biblioteca BFT. Esta ordenação deve ser mantida, de modo a evitar inconsistências entre as várias réplicas. Segundo, todas as operações de escrita relativas a um recurso têm de ser executadas isoladamente de todas as outras operações. Por exemplo, se existir um cliente a fazer uma escrita num ficheiro, não poderá haver outro cliente a fazer qualquer operação, seja de escrita ou de leitura, sobre esse ficheiro. No entanto, nada impede que sejam efectuadas leituras concorrentes sobre o mesmo ficheiro ou escritas concorrentes em ficheiros distintos.

```
1 public Result operation(String resource){
2
3   Lock lock = new Lock(resource, order, threadId);
4   List lockList = new ArrayList<Lock>();
5   lockList.add(lock);
6   ContentionManager.acquirelocks(lockList);
7   doOperations;
8   ContentionManager.releaseLocks(lockList);
9 }
```

Figura 3.4: Exemplo de um método onde é utilizado o `ContentionManager`.

A reserva de um recurso é representada pela classe abstracta `Lock`, como podemos ver na Figura 3.5. Esta pode ser estendida de modo a representar um acesso a qualquer recurso, independentemente da aplicação, sendo composta por uma `String` que identifica o recurso, como o caminho absoluto de um ficheiro, a ordem da operação imposta pela biblioteca BFT e o identificador da *thread* que está a executar a operação. Uma reserva pode ainda ter vários significados, dependendo do objectivo da operação, podendo representar uma operação de leitura ou de escrita.

Assim, o método `acquireLocks(List<Lock> l)` apenas retorna quando obtiver permissão para tal. Internamente ao mecanismo de contenção, é verificado se a operação pode ser executada ou se, caso contrário, tem de aguardar por permissão. A operação poderá ser executada quando não houver nenhum conflito com outras operações, ou seja, quando todos os *locks* presentes na lista recebida como argumento estiverem satisfeitos. É ainda necessário verificar que a ordem imposta pela biblioteca BFT é respeitada. Neste último caso é necessário garantir que cada *lock*, relativo a um recurso, é obtido pela ordem correcta, atribuída pela biblioteca BFT. Por exemplo, no caso de existirem dois *locks*, relativos ao mesmo recurso, aquele que tiver o número de ordem mais elevado terá de aguardar até que o *lock* anterior seja libertado. Note-se que esta condição deve verificar-se independentemente da ordem em que estes dois *locks* dêem entrada no mecanismo de contenção, de modo a evitar alterações à ordem dos pedidos. Na Secção 4.4 documento

iremos apresentar mais detalhes acerca desta solução.

```
package util;

public abstract class Lock {
  private int bftOrder;
  private long threadId;
  private String id;

  public Lock(String resource, int order, long threadId) {
    this.id = resource;
    this.bftOrder = order;
    this.threadId = threadId;
  }

  /*
   * Getters & Setters
   */

}
```

Figura 3.5: Classe abstracta Lock.

3.3 Caso experimental

Um dos requisitos dos algoritmos de tolerância a falhas bizantinas é o facto de as aplicações terem de ser baseadas em operações deterministas, isto é, dado um estado consistente do sistema, a execução de uma operação leva sempre a um mesmo estado, também ele consistente, e retorna sempre o mesmo resultado.

Os sistemas de ficheiros distribuídos, a par das bases de dados distribuídas, surgem assim como uma aplicação ideal para testar este tipo de solução. Radwin et al. [Rad97] criou o JNFS (Java Network FileSystem), um sistema de ficheiros distribuído implementado na linguagem Java, utilizando o RMI como mecanismo de comunicação entre os componentes do sistema. Desta forma, a arquitectura base da aplicação é baseada num modelo cliente/servidor, sendo que este último corre directamente sobre o sistema de ficheiros.

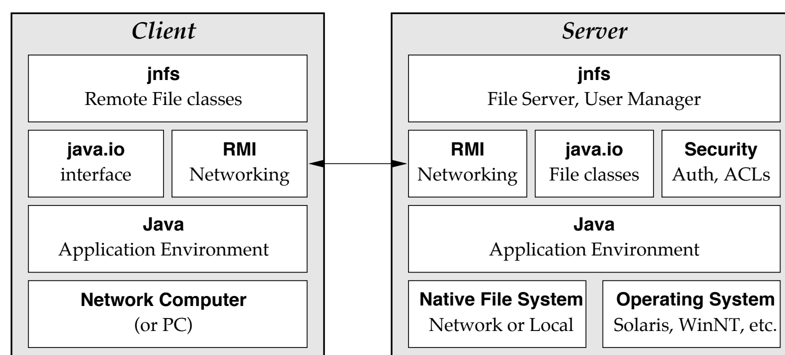

Figura 3.6: Arquitectura do JNFS. Retirado de [Rad97].

Como é possível observar na Figura 3.6, o sistema é composto por um cliente, que irá efectuar pedidos RMI a um servidor, responsável por executar as operações. Através do JNFS é possível efectuar a generalidade das operações mais comuns em sistemas de ficheiros, como criar ou remover ficheiros, escritas e leituras de ficheiros, criar ou remover directorias, entre outras. O cliente tem ao seu dispor um conjunto de classes que estendem algumas classes do pacote `java.io` responsáveis pela manipulação de ficheiros remotos, com o intuito de facilitar a programação de aplicações.

Do lado do servidor, existe uma classe principal `FileServer` que permite fazer toda a gestão de utilizadores e de ficheiros. Esta classe implementa várias interfaces, cada uma com objectivos distintos, fornecendo vários tipos de operações. Uma das principais funções do servidor é servir como fábrica de objectos de ficheiros remotos, que permitem ao cliente efectuar operações nos ficheiros.

No caso de ser utilizado um algoritmo de tolerância a falhas bizantinas, a arquitectura do sistema irá, forçosamente, alterar-se. Como é visível na imagem 3.7, passamos assim a ter vários servidores, que representam o sistema de ficheiros e onde serão efectuadas as operações. A comunicação entre cliente e servidores passa a ser efectuada através da biblioteca BFT integrada na camada de transporte do RMI, que propaga cada pedido para os servidores e devolve o resultado final ao cliente. Através desta biblioteca, são mascaradas possíveis falhas bizantinas dos servidores. Note-se ainda que, do ponto de vista do cliente, a arquitectura mantém-se, uma vez que estas alterações são transparentes.

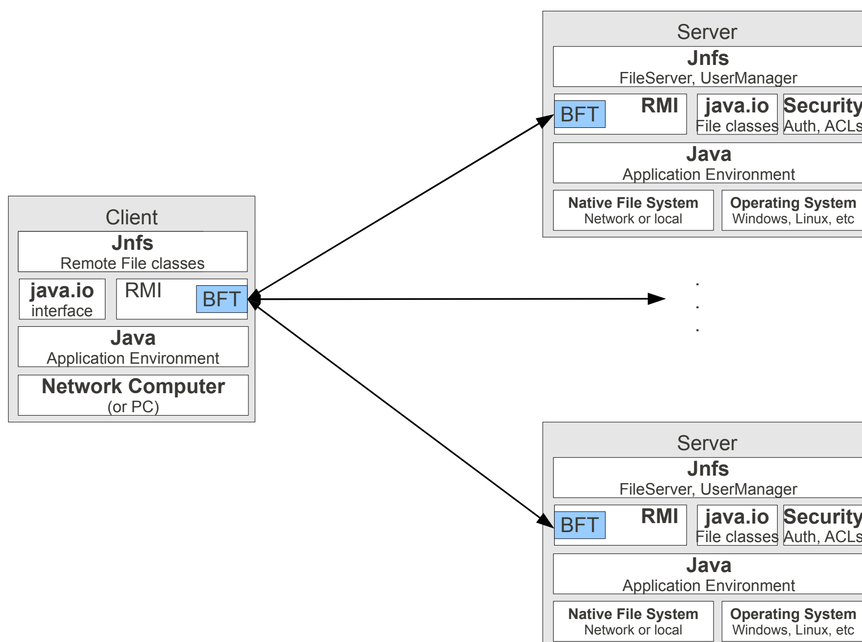

Figura 3.7: Arquitectura do JNFS, no caso de ser utilizado BFT.

4

Implementação

Neste capítulo apresenta-se a implementação do modelo descrito anteriormente, abordando alguns detalhes importantes do nosso trabalho. Este encontra-se dividido em duas fases distintas. Primeiro, uma implementação de RMI baseada em UDP e, numa segunda fase, uma versão que tem por meio de comunicação um algoritmo com tolerância a falhas bizantinas.

4.1 RMI

O nosso trabalho visa implementar um mecanismo de tolerância a falhas bizantinas em aplicações distribuídas baseadas no mecanismo de invocação remota RMI. Para tal, é necessário introduzir replicação nestas aplicações, passando a executar com vários servidores em vez de apenas um.

Uma vez que o RMI se encontra implementado em várias camadas, apenas necessitamos alterar a implementação da camada de transporte. Note-se que o termo camada é utilizado de um ponto de vista conceptual, representando um conjunto de classes onde se encontram implementados os serviços.

De modo a compreender a nossa implementação, é importante primeiro conhecer alguns detalhes da forma como o RMI funciona e como a camada de transporte é utilizada.

A camada de transporte encontra-se no nível mais baixo do RMI e tem como função efectuar as comunicações entre cliente e servidor. Estas comunicações são efectuadas através de um fluxo de mensagens, um *stream*, que é composto maioritariamente pelas invocações efectuadas pelo cliente e pelos respectivos resultados devolvidos pelo servidor. É importante frisar que este mecanismo apenas está pensado para comunicações do tipo 'um para um', ou seja, *unicast*, o que levanta alguns problemas quando o objectivo é

estabelecer comunicações 'um para muitos'. Este problema será abordado na secção 4.3.

Existem algumas componentes desta camada que merecem uma atenção especial, uma vez que são aqueles que implementam os serviços necessários e que podem ser redefinidos de modo a alterar o padrão de comunicação.

- **`Socket`** – Esta classe representa um canal de comunicação entre dois objectos. Tendo associado um `InputStream` e um `OutputStream`.
- **`InputStream`** – Representa o ponto de entrada dos dados no *socket* .
- **`OutputStream`** – Representa o ponto de saída dos dados no *socket*.
- **`ServerSocket`** – Representa um *socket* utilizado pelo servidor. Este vai ser responsável pela criação de novos canais de comunicação para cada cliente.
- **`RMIClientSocketFactory`** – Interface que define uma fábrica de *sockets* para o cliente.
- **`RMIServerSocketFactory`** – Interface que define uma fábrica de *sockets* para o servidor.

No momento em que um cliente procura obter uma ligação para um servidor remoto, é necessário estabelecer um canal de comunicação entre eles. Este canal, criado pela camada de transporte, é visto como um *socket*, onde são introduzidas mensagens por um dos intervenientes, de modo a serem lidas pelo outro. Como podemos observar na Figura 4.1, um *socket* pode ser visto como um canal que liga dois pontos na rede, sendo que em cada um deles existe um ponto de saída de mensagens, conhecido como *OutputStream*, e um ponto de entrada, denominado *InputStream*. Podemos dizer que o *OutputStream* de um cliente se encontra ligado ao *InputStream* do servidor, e vice-versa, criando um canal de comunicação entre eles.

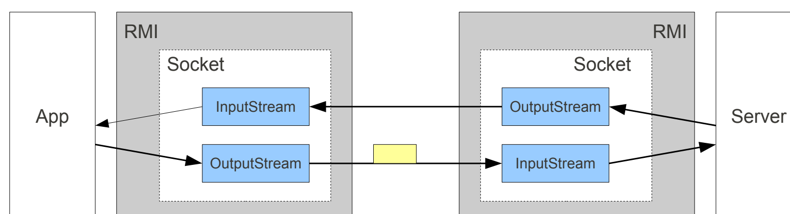

Figura 4.1: Exemplo de um *socket* RMI.

Na implementação actual do RMI estes *sockets* são criados sobre conexões TCP/IP no momento em que se estabelece a ligação entre cliente e servidor, naquilo a que se chamam as fábricas de *sockets*. Estas fábricas podem ser substituídas por outras que criem outro tipo de *sockets* com outras características. Existe uma destas fábricas relativa ao cliente, definida pela interface `RMIClientSocketFactory`, e outra para o servidor, definida por `RMIServerSocketFactory`. Assim, podemos criar novas fábricas, como podemos

observar no exemplo da Figura 4.2, que devolvem uma nova implementação de `Socket`, beneficiando do mecanismo de herança do Java, alterando o padrão de comunicação de forma transparente para a aplicação.

```
1
2  public class MyClientSocketFactory implements RMIClientSocketFactory{
3
4    public Socket createSocket(String host, int port) throws IOException {
5      return new MySocket( host, port);
6    }
7  }
```

Figura 4.2: Exemplo de uma fábrica de *sockets* do cliente.

As fábricas de *sockets* têm funções um pouco diferentes no cliente e no servidor. Enquanto que no cliente é na fábrica que são criados todos os *sockets*, no servidor é um pouco diferente, como se pode observar na Figura 4.3. Uma vez que as interacções são sempre iniciadas pelo cliente, do lado do servidor existe ainda outro componente responsável por atender a novos pedidos de conexão por parte de cliente, actuando como um *listener*. Este serviço é implementado pela classe `ServerSocket` que trata todos os novos pedidos de conexão, por parte do cliente criando um novo *socket* que representa a ligação entre o servidor e esse cliente específico. Assim, a fábrica de *sockets* do servidor apenas cria um destes objectos para tratar das novas conexões.

O RMI, ao receber um novo pedido de conexão por parte do cliente invoca o método `accept()` da classe `ServerSocket`. Este método devolve o *socket* que ficará associado ao cliente e que contém as respectivas *InputStream* e *OutputStream*.

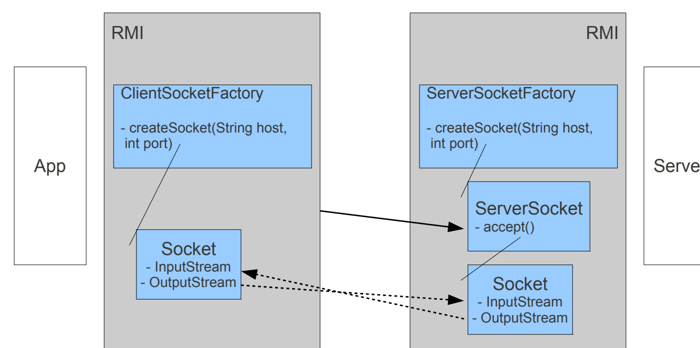

Figura 4.3: Esquema de uma conexão RMI.

4.1.1 Protocolo RMI

O RMI é suportado na sua base por um protocolo de comunicação, definido em [RMI]. Este protocolo é composto por uma gramática que determina a forma como dois objectos devem comunicar entre si e o formato que cada mensagem deve ter.

Uma interacção entre dois objectos pode ser vista como um *stream* em que o cliente e o servidor comunicam através de um canal de comunicação.

O *stream* inicia-se com uma troca de mensagens que tem como objectivo o estabelecimento da conexão entre os dois objectos. Nestas primeiras mensagens, os objectos trocam entre si os seus identificadores únicos, atribuídos pela JVM respectiva No entanto, grande parte das interacções são compostas por invocações de métodos e os seus respectivos resultados. O primeiro byte de cada mensagem indica o seu tipo, por exemplo uma mensagem cujo primeiro byte seja `0x50` representa uma invocação de um método, enquanto que o byte `0x51` indica o retorno de um resultado.

As mensagens de invocação e retorno são compostas por uma estrutura semelhante. Enquanto que o início destas mensagens é ocupado com o identificador único do objecto para o qual se destina a mensagem, de modo a que a JVM possa encaminhar a mensagem, o restante é ocupado com conteúdos diferentes. No caso das invocações, a mensagem contém os argumentos do método. Já nas mensagens de retorno, o restante da mensagem é ocupado com o valor de retorno do método. Note-se que, em ambos os casos, os dados são codificados através do mecanismo de serialização do Java.

Na Figura 4.4 podemos ver a composição de uma invocação. Podemos ver que primeiro é indicado o tipo da mensagem, seguido o identificador do destinatário. Este, é gerado pela máquina virtual e é composto, entre outros dados, por um *timestamp*. O resto da mensagem é ocupada com os detalhes da invocação, como o método e os argumentos.

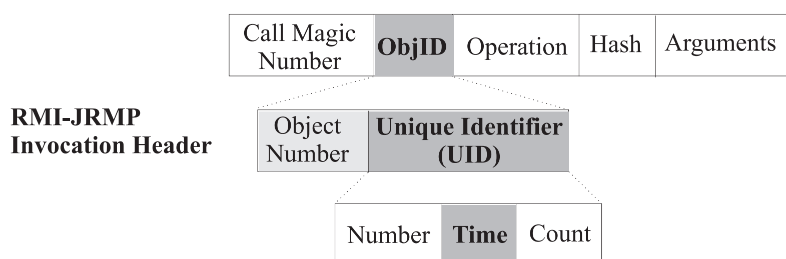

Figura 4.4: Composição da mensagem de uma invocação RMI.

É importante ainda frisar a existência de outros dois tipos de mensagem. As mensagens de *ping*, cujo objectivo é testar se uma máquina virtual remota ainda se encontra activa, e as mensagens de *DgcAck*, que fazem parte do mecanismo de *garbage colection* distribuída, que verifica se ainda existem referências em máquinas virtuais remotas para um determinado objecto. Na Figura 4.5 encontra-se uma listagem dos principais tipos de mensagens, e respectivos códigos, que compõem um stream de comunicação RMI.

Tipo	Código
invocação	0x50
retorno	0x51
ping	0x52
pingAck	0x53
DgcAck	0x54

Figura 4.5: Tipos de mensagens RMI.

4.2 RMI sobre UDP

Numa primeira fase do nosso trabalho optámos por fazer uma implementação da camada de transporte do RMI através do protocolo UDP. Esta opção deveu-se ao facto de o modelo de comunicação deste protocolo, do ponto de vista da aplicação, ser semelhante aquele que se verifica com a biblioteca BFT.

O UDP é um protocolo de comunicação da camada de transporte e baseia-se na simples troca de pacotes entre dois intervenientes numa ligação efectuada através de uma rede. Relativamente ao TCP/IP, é um protocolo que oferece poucas garantias, uma vez que não garante a ordenação dos pacotes ou mesmo a sua entrega ao seu destinatário. É, por isso, um protocolo bastante mais leve e, potencialmente, mais eficiente do que o TCP, utilizado originalmente pelo RMI. Esta diferença nas características dos dois protocolos leva a que tenham usos distintos, em aplicações distintas, dependendo das restrições das mesmas.

É importante referir que o RMI funciona através da troca de mensagens, vistas como uma *stream*, entre dois objectos Java, em que cada uma delas pode representar, por exemplo, uma invocação de um método ou o retorno de um valor. Assim, cada pacote UDP será composto por esta mensagem, que na prática é representada por um conjunto de *bytes*.

Para fazer com que o RMI passasse a executar segundo este protocolo, criámos as seguintes classes para substituir aquelas que compõem a implementação de raiz da camada de transporte:

- **`UdpSocket`** – Esta classe implementa um canal de comunicação UDP entre o cliente e o servidor. É composto por um `InputStream` e um `OutputStream`.

- **`UdpInputStream`** – Representa o ponto de entrada dos dados no *socket*, ou seja, é através desta classe que são recebidos os pacotes enviados pelo servidor, no caso do cliente.

- **`UdpOutputStream`** – Representa o ponto de saída dos dados no *socket*, ou seja, é através desta classe que são enviados os pacotes contendo as mensagens do cliente para o servidor e vice-versa.

- **UdpServerSocket** – Representa um *socket* utilizado pelo servidor. Este vai ser responsável pela criação de novos canais de comunicação para cada cliente, tal como pela recepção dos pacotes enviados pelo cliente.
- **UdpClientSocketFactory** – Interface que define uma fábrica de *sockets* para o cliente criando, neste caso, instâncias de `UdpSocket`.
- **UdpServerSocketFactory** – Interface que define uma fábrica de *sockets* para o servidor criando, neste caso, instâncias de `UdpServerSocket`.

Na nossa aproximação, o fluxo de mensagens entre o cliente e o servidor é propagado recorrendo a mensagens UDP. Ao contrário do protocolo TCP, no protocolo UDP não existe a noção de conexão, pelo que é necessário emular uma conexão usando o protocolo UDP. Para tal, o servidor mantém informação das conexões virtuais existentes, usando para esse efeito o endereço e porta do cliente. Esta informação é mantida por um objecto `UdpServerSocket`, que também é responsável por criar novas conexões, numa tabela de clientes, onde são associados os endereços de cada cliente a uma instância de `UdpInputStream`. Como podemos observar na Figura 4.6, os pacotes enviados por um cliente através da sua `UdpOutputStream` são primeiro recebidos por `UdpServerSocket` que os encaminha para o servidor, através da instância guardada na tabela de `UdpInputStream` referente ao cliente que enviou a mensagem.

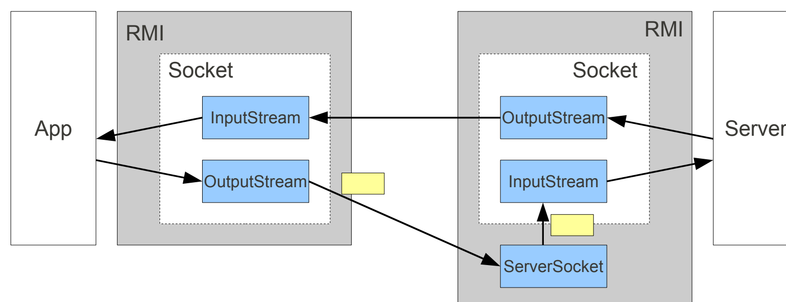

Figura 4.6: Exemplo de comunicação entre dois objectos RMI, utilizando `UdpSocket`.

Assim, as comunicações passam a ser executadas através de um mecanismo simples de comunicação da linguagem Java, baseado na classe `DatagramSocket`. Esta permite criar uma ligação entre dois objectos que comunicam através do envio de pacotes. Como podemos ver na Figura 4.7, uma mensagem, que não é mais do que um `byte[]`, é encapsulada num pacote UDP, através da classe `DatagramPacket` e enviado para o receptor, seja ele o cliente ou o servidor. Há que salientar que alguns pormenores, como outros métodos auxiliares, se encontram omitidos, de forma a manter a simplicidade do código apresentado . Por exemplo, no caso de uma mensagem ter um tamanho maior que um pacote (1500 bytes) será divida em vários pacotes que serão enviados para o receptor um a um.

Na Figura 4.8 podemos observar de que forma é recebida uma mensagem pelo servidor, novamente através da classe `DatagramSocket`, com a diferença que desta vez é

```
1
2  package udpsocket;
3
4  public class UdpOutputStream extends OutputStream {
5
6    private SocketAddress remoteAddr;
7    private DatagramSocket socket;
8    private DatagramPacket packet;
9
10   /*
11    * Constructor and other methods
12    */
13
14   public void write(byte[] msg, int off, int len) {
15     packet = new DatagramPacket(msg, off,  len, remoteAddr);
16     socket.send(packet);
17   }
18 }
```

Figura 4.7: Excerto da classe `UdpOutputStream`.

utilizado o método `receive(DatagramPacket packet)`. A classe `Listener` tem associada uma *thread*, a executar do lado do servidor, que é responsável por receber e tratar os pacotes vindos dos clientes. Esta é composta por três componentes principais: (1) udpSocket, para receber novos pacotes vindos dos clientes; (2) `clientsUdpIn`, que representa uma tabela onde para cada cliente, identificado pelo seu endereço e porta, está associado o seu `UdpInputStream`; (3) `newClients`, que representa uma lista onde são colocadas as novas conexões existentes antes de ser criado o respectivo *socket*.

Para estabelecer uma ligação entre cliente e servidor, temos então de passar por duas fases que são executadas concorrentemente. Na classe `Listener` temos conhecimento de uma nova ligação de duas formas: (1) quando é recebido um pacote com uma mensagem que indica que um novo cliente deseja estabelecer uma nova ligação; (2) quando o método `accept()` é executado. A necessidade destas duas fases explica-se pelo facto de quando o método `accept()` é chamado não termos acesso ao endereço do cliente o que tornaria impossível associar-lhe o *UdpInputStream*. Assim, ainda antes de ser criado o *socket*, o cliente envia uma mensagem UDP a assinalar o início de uma nova conexão de modo a que o servidor fique a conhecer o seu endereço e associar-lhe logo o seu *UdpInputStream*, que fará parte do *socket* de comunicação.

Na Figura 4.9 podemos encontrar a demonstração deste mecanismo. Quando chega um pacote correspondente a uma nova conexão é introduzido o endereço do cliente numa lista de novos clientes (`newClients`) e criada uma nova `UdpInputStream` associada ao endereço do cliente, composto por endereço IP e porta, na tabela de clientes. No momento em que o método `accept()` é chamado, o endereço é removido da lista e criado o novo `UdpSocket`, contendo a `UdpInputStream` criada anteriormente, por onde serão enviados para o servidor os dados que sejam recebidos deste cliente. O *socket* só pode ser criado após ser recebida a primeira mensagem do cliente, pelo que, no método

```
package udpsocket;

class Listener extends Thread{

  //    socket for incoming packets
  private DatagramSocket udpSocket;
  //    table do keep track of all the connections
  private Map<SocketAddress, UdpInputStream> clientsUdpIn;
  //    list of new clients
  private List<SocketAddress> newClients;

  //      Constructor and other methods

  public void run() {
    // Here, the server will listen for new requests
    while(true){
        try {
          DatagramPacket packet = new DatagramPacket(buf, buf.length);
          udpSocket.receive(packet);
          //handle packet
          } catch (IOException e  ) {
              e.printStackTrace();
          }
  }
}
```

Figura 4.8: Excerto da classe `Listener`.

`accept()` é necessário esperar por esta mensagem caso não exista nenhuma. Note-se que esta mensagem não será passada para a aplicação, sendo apenas interpretada pela *thread* de atendimento de pedidos e depois descartada.

Durante o funcionamento normal da aplicação as mensagens que correspondem às invocações efectuadas pelo cliente são enviadas através de pacotes UDP e recebidas pelo servidor. Este é responsável por colocar o conteúdo de cada mensagem no *InputStream* respectivo, que se encontra no mapa `clientsUdpIn`, onde as mensagens serão armazenadas até serem consumidas pela aplicação. Por último, no fim da conexão o *InputStream* é removido do mapa, de modo a libertar espaço e a tornar possíveis novas conexões.

4.3 RMI com tolerância a falhas bizantinas

Um dos objectivos do nosso trabalho foi implementar um mecanismo de tolerância a falhas bizantinas no RMI do Java e avaliar o impacto que esta alteração tem no desempenho das aplicações. Para isso, como foi descrito nas secções anteriores, foi substituída a camada de transporte por uma nova versão que permite tolerar falhas de cariz arbitrário.

4.3.1 Biblioteca BFT

De modo a que as operações sejam executadas através de um algoritmo de tolerância a falhas bizantinas utilizamos a biblioteca de comunicação do projecto Byzantium [PRHL08].

```
package udpsocket;

class Listener extends Thread{

  //    Constructor and other methods

  //    handle packet
  if(isEqual(data,UdpSocket.FIRST_MESSAGE)){
      UdpInputStream udpIn = new UdpInputStream(udpSocket, true );
      clientsUdpIn.put(address,UdpIn);
      synchronized( newClients) {
        newClients.add(address);
        newClients.notifyAll();
      }
  } else
    //  Write message to server's InputStream
    clientsUdpIn.get(address).write(packet.getData(), 0, packet.getLength());
}

  public Socket accept() throws IOException {
    synchronized( newClients) {
      while (newClients.size() == 0)
        try {
          // wait for a new client..
          newClients.wait();
        } catch (InterruptedException e) {
          // do nothing
        }
        SocketAddress addr = newClients.remove(0);
        return new UdpSocket( udpSocket, addr, clientsUdpIn.get(addr));
    }
  }
}
```

Figura 4.9: Excerto da classe `Listener`.

Esta biblioteca baseia-se no algoritmo PBFT [CL02] e, como tal, necessita de $3f + 1$ servidores, conhecidos como réplicas, para executar as operações, sendo f o número de réplicas que podem exibir comportamentos arbitrários. Entre as réplicas existe sempre uma que é considerada a primária, à qual os clientes efectuam os pedidos que serão propagados para as restantes réplicas, conhecidas como secundárias. Durante esta propagação de pedidos, é necessário que sejam efectuadas três rondas de comunicações entre as réplicas, de forma a que estas cheguem a um consenso acerca da ordem em que todas devem executar o pedido, uma vez que estes necessitam de uma ordenação total.

A implementação deste algoritmo é feita com base em duas classes principais:

- **`PBFTClient`** – Esta classe permite que uma aplicação submeta pedidos para serem executados nos servidores. No fim desta execução, os servidores enviam o resultado ao cliente que verifica se os resultados são todos consistentes entre eles.

- **`PBFTServer`** – Esta classe representa um servidor PBFT. Existem dois modos de execução possíveis, primário e secundário. Um servidor primário tem funções acrescidas relativamente aos secundários, sendo responsável por receber os pedidos enviados pelo cliente e propagá-los para as restantes réplicas. Todas elas, primárias e secundárias, executam a operação e enviam o seu resultado para o cliente.

O uso desta biblioteca representa uma camada intermédia entre cliente e servidor. Tal como podemos observar na Figura 4.10, em cada uma das réplicas que compõem o sistema, existe uma instância de `PBFTServer`, seja primária ou secundária. Já no cliente, existe uma instância de `PBFTClient`.

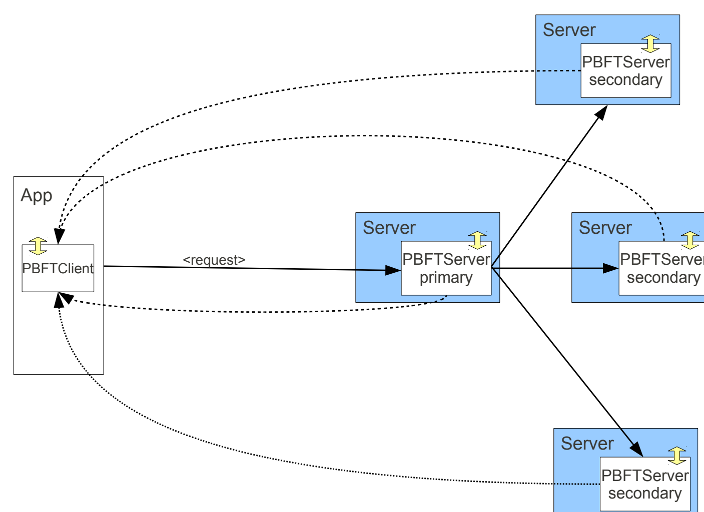

Figura 4.10: Arquitectura de uma aplicação, utilizando a biblioteca BFT.

Existem outras componentes que compõem esta biblioteca que podem ser importantes para compreender alguns detalhes do trabalho.

- **`RSOperation`** – Esta interface define uma mensagem que representa uma operação no âmbito da biblioteca. Os objectos que implementam esta interface, que podem representar qualquer tipo de operação, são enviados pelo cliente para as réplicas onde serão tratadas conforme a aplicação.
- **`RSResult`** – Esta interface define uma mensagem que representa um resultado de uma operação no âmbito da biblioteca. Os objectos que implementam esta interface, que podem representar qualquer tipo de resultado, são enviados pelos servidores para o cliente, que é responsável pela sua validação. Um resultado é considerado válido quando existirem $f + 1$ valores consistentes.
- **`PBFTServerOp`** – Esta classe representa uma operação no âmbito da biblioteca e contém a mensagem que contém os dados da operação, tal como o endereço do cliente que fez o pedido.

Com base nestes componentes esta biblioteca tem um comportamento um pouco diferente do habitual. Em vez de as operações serem executadas de uma forma síncrona, em que o cliente fica bloqueado à espera do seu resultado, os pedidos são enviados e executados nos servidores assíncronamente. Assim, a biblioteca apresenta a seguinte interface: getResult()

- **`executeBFToperation(RSOperation operation)`** – Método da classe `PBFTClient` que permite que o cliente envie uma operação para ser executada nos servidores. Este método retorna um objecto da classe `PBFTResult`.
- **`getNextOperation()`** – Método da classe `PBFTServer` que permite que o servidor peça à biblioteca a próxima operação a ser executada. Este método bloqueia enquanto não existirem mais operações a executar.
- **`returnResult(RSResult result)`** – Método da classe `PBFTServerOp` devolva à biblioteca o resultado final de uma operação. Este resultado será posteriormente entregue ao cliente pela biblioteca.
- **`getResult()`** – Método da classe `PBFTResult` que permite que um cliente faça um pedido à biblioteca pelo resultado de uma operação. Este método bloqueia enquanto o resultado não estiver disponível.

É importante referir que a biblioteca de BFT utilizada, permitindo a execução concorrente de operações no servidor, apresenta uma limitação importante: não implementa o mecanismo de *checkpointing*. Este mecanismo é usado essencialmente para permitir às réplicas descartar as mensagens antigas sem que isso impeça uma réplica que não observou algumas mensagens recuperar o seu estado. Assim, a aplicação não necessita de implementar as operações usuais relativas a esta funcionalidade.

4.3.2 Implementação

Como vimos na secção anterior, a biblioteca utilizada é vista como uma camada intermédia entre cliente e servidor. No caso do RMI, a biblioteca vai situar-se na camada de transporte, ou seja, é através dela que serão feitas as comunicações entre cliente e servidores.

Desta forma, criámos as seguintes classes para substituir aquelas que compõem a implementação de raiz:

- **`BFTSocket`** – Esta classe representa um canal de comunicação entre o cliente e os servidores. É composto por um `InputStream` e um `OutputStream`.
- **`BFTInputStream`** – Representa o ponto de entrada dos dados no *socket*.
- **`BFTOutputStream`** – Representa o ponto de saída dos dados no *socket*. Aqui, as mensagens são enviadas para as réplicas através do cliente (`PBFTClient`).
- **`BFTServerSocket`** – Representa um *socket* utilizado pelo servidor. Este vai ser responsável pela criação de novos canais de comunicação para cada cliente. Neste caso, é aqui que podemos encontrar uma instância de um `PBFTServer` a correr, de modo a atender os pedidos do cliente.
- **`BFTClientSocketFactory`** – Interface que define uma fábrica de *sockets* para o cliente criando, neste caso, instâncias de `BFTSocket`.
- **`BFTServerSocketFactory`** – Interface que define uma fábrica de *sockets* para o servidor criando, neste caso, instâncias de `BFTServerSocket`.

Como podemos observar na Figura 4.11, a arquitectura do sistema sofre algumas alterações. Passamos então a ter vários servidores replicados, ao todo $3f + 1$, em vez de apenas um. Todos estes servidores, ou réplicas, passam a executar as operações pela mesma ordem, imposta pela biblioteca de comunicação BFT, e a retornar os resultados directamente ao cliente, que é responsável pela sua validação.

O RMI passa agora a enviar as mensagens através da biblioteca BFT descrita na secção anterior. O *socket* do cliente passa a enviar os pedidos para os servidores através de uma instância de `PBFTClient`, enquanto que, do lado de cada servidor, passa a existir uma instância de `PBFTServer`, que será responsável por receber as mensagens do cliente e passá-las para a aplicação.

Por comparação à implementação anterior, feita através de UDP, e como já foi referido, a semântica de comunicação entre cliente e servidor é semelhante. Cada mensagem é encapsulada num objecto `RMIOp`,implementando a interface `RSOperation`, juntamente com o endereço do cliente e enviada através da instância de `PBFTClient` para os servidores. Na Figura 4.12 é apresentada a forma como é feito o envio de uma mensagem por parte do cliente no seu `BFTOutputStream` através de `PBFTClient`.

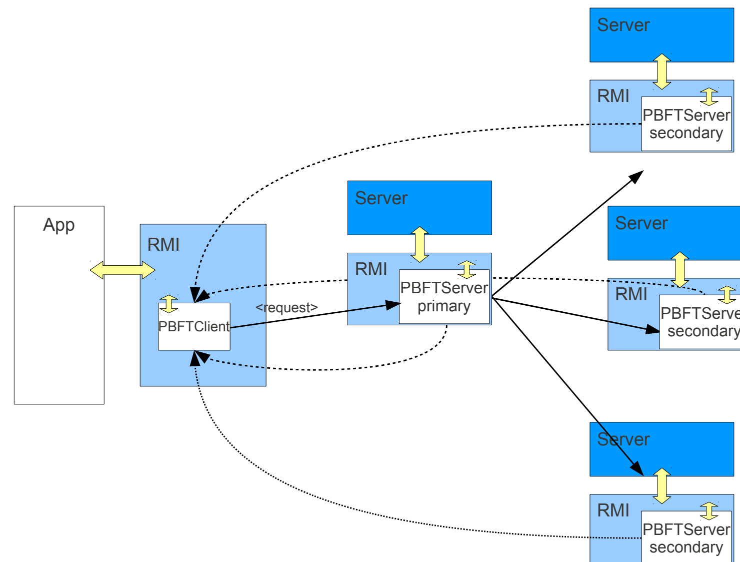

Figura 4.11: Arquitectura de uma aplicação RMI, utilizando a biblioteca BFT.

```
1  package bftsocket;
2
3  public class BFTOutputStream extends OutputStream {
4
5    private SocketAddress remoteAddr;
6    private PBFTClient pbftClient;
7
8    /*
9     * Constructor and other methods
10    */
11
12   public void write(byte[] msg, int off, int len) {
13     RSOperation op = new RMIOp(msg, msg.length, remoteAddr, clientPort);
14     PBFTResult rs = pbftClient.executeBFToperation(op);
15   }
16 }
```

Figura 4.12: Excerto da classe `BFTOutputStream`.

A forma como um servidor recebe uma mensagem também é semelhante à forma como apresentámos na versão UDP. Existe uma *thread* de atendimento de pedidos, novamente implementada pela classe `Listener`, que fica bloqueada à espera de uma nova mensagem invocando o método `getNextOperation()` do objecto `pbftServer`. Note-se que esta mensagem é recebida pelas réplicas após ser ordenada num processo interno à biblioteca BFT e, como tal, a mensagem pode ser logo transmitida à aplicação através do `BFTInputStream` respectivo. O processo de criação de novos *sockets* é semelhante à versão anterior, baseada em UDP. Assim, o cliente envia uma mensagem para o servidor indicativa de uma nova conexão antes que seja invocado o método `accept()`, de modo a que se possa criar previamente o `BFTSocket` respectivo ao cliente.

```
1  package bftsocket;
2
3  class Listener extends Thread{
4
5    //    instance of PBFTServer to receive requests
6    private PBFTServer pbftServer;
7    //    table do keep track of all the connections
8    private Map<SocketAddress, Socket> clientsIn;
9    //    list of new clients
10   private List<SocketAddress> newClients;
11
12   //    Constructor and other methods
13
14   public void run() {
15     // Here, the server will listen for new requests
16     while(true){
17         try {
18           PBFTServerOp pbftOp = pbftServer.getNextOperation() ;
19           RSOperation op = pbftOp.getOperation();
20           //handle op
21           } catch (IOException e  ) {
22               e.printStackTrace();
23           }
24   }
25 }
```

Figura 4.13: Excerto da classe `Listener`.

No final da execução da operação, o servidor envia, através do *outputstream* o resultado para o cliente. Este envio deve ser feito através do objecto que representa a operação dentro do sistema (`PBFTServerOp`). Uma vez que este objecto é obtido no momento em que se recebe a operação, ou seja, na classe `Listener`, terá de ser passado para o *outputstream*. Note-se que o resultado é, mais uma vez, encapsulado num objecto antes de ser enviado para o cliente. Este objecto, representado pela classe `RMIResult`, contém apenas o `byte[]` e implementa a interface `RSResult`, que define um resultado no âmbito da biblioteca BFT.

Como já foi dito neste documento, na Secção 4.1, uma mensagem RMI é composta por um `byte[]` que passa primeiro pela camada de transporte antes de ser interpretada pela JVM e entregue ao objecto Java ao qual se destina. Dentro da mensagem encontra-se o

identificador único do objecto de modo a que a JVM possa fazer o encaminhamento. Esta solução está pensada para comunicações 'um para um' o que gera problemas no caso de estarmos a trabalhar com comunicações 'um para muitos', como é o caso.

Assim, todas as mensagens têm apenas um destinatário, seja ele o cliente ou o servidor, que está explícito no cabeçalho da mensagem, como explicámos na Secção 4.1.1. Este destinatário é identificado pelo identificador único que lhe é atribuído pela máquina virtual na altura da sua criação, pelo que não pode ser alterado.

Tendo em conta que o cliente julga que efectua todas as comunicações com um único servidor, esta vai ser a única referência que vai obter e, como tal, a mensagem tem como destinatário o objecto deste servidor. Uma vez que todos os servidores têm identificadores diferentes, quando a mensagem é propagada por aquela réplica para as restantes, as respectivas JVM não têm conhecimento de nenhum objecto cujo identificador único corresponda aquele que se encontra na mensagem, o que gera uma excepção. A solução passa por cada réplica guardar em memória o seu próprio identificador, que é possível descobrir através da serialização do próprio objecto, e substitui-lo nas mensagens que recebe, e que correspondam a invocações. Note-se que os identificadores têm sempre o mesmo tamanho e ocupam sempre uma secção definida no cabeçalho da mensagem.

As mensagens de retorno contêm ainda o identificador do servidor pelo que, apesar de conterem resultados idênticos, as mensagens de todas as réplicas serão diferentes e, como tal, rejeitadas pelo cliente. A solução para este problema passa por fazer a troca deste identificador, pelo identificador de uma réplica primária. Para tal, no início da execução do sistema a réplica primária do momento envia para as réplicas secundárias o seu identificador, que ficará guardado em memória.

Já foi referido anteriormente que a biblioteca BFT permite a execução concorrente de operações nos servidores. Este comportamento é obtido através da operação `getNextOperation()` da classe `PBFTServer` que retorna a próxima operação a ser executada. Assim, uma vez que na classe `Listener` este método é invocado consecutivamente colocando a mensagem relativa à operação no *InputStream* respectivo, sem que seja devolvido um resultado para a operação, as operações são executadas concorrentemente pela aplicação.

Uma forma de evitar este comportamento e impor a sequencialidade das operações é obrigar a que apenas seja invocado o método `getNextOperation()` após ser retornado o resultado da operação corrente. Para tal, o servidor aguarda que este resultado seja devolvido ao cliente, através do *OutputStream*. Este último é assim responsável por notificar a classe `Listener` que a operação corrente terminou e que já poderá pedir à biblioteca a operação seguinte.

4.4 Mecanismo de Locks

Na Secção 3.2.3 introduzimos um mecanismo de gestão de recursos geral, que pode ser adaptado para ser utilizado em conjunto com várias aplicações. Nesta secção iremos apresentar com maior detalhe a sua implementação, tal como a forma como o utilizámos

no âmbito do sistema de ficheiros JNFS.

Tratando-se de um sistema de ficheiros, um recurso pode ser visto como um ficheiro ou uma directoria, podendo ser criado, removido, lido ou alterado. Assim, as operações podem ser agrupadas em operações de leitura e de escrita. Para representar estas operações e a forma como o nosso mecanismo de contenção se deveria comportar, criámos as seguintes classes, que estendem a classe `Lock`, já descrita anteriormente:

- **`ReadLock`** – Deve ser utilizada em operações apenas de leitura, como ler o conteúdo de um ficheiro. Quando dá entrada no `ContentionManager` poderá executar concorrentemente com outras operações de leitura, desde que não haja nenhuma operação de escrita em espera.

- **`WriteLock`** – Deve ser utilizada em operações que alterem os dados, como criar uma directoria ou escrever num ficheiro.

Como foi dito anteriormente na Secção 3.2.3, todas as operações de escrita devem executar isoladamente, não entrando em conflito com outras operações, sejam elas de escrita ou de leitura. Assim, uma operação de escrita deverá procurar obter uma reserva, ou *lock*, de escrita antes de executar. Podemos observar na Figura 4.14 que antes de a operação `writeOperation`, que efectua alguma operação de escrita no ficheiro designado pelo argumento `path`, ser executada deve ser criado um objecto `WriteLock`, que irá conter o caminho para o ficheiro e o identificador da *thread* que está a executar o método, seguido da execução do método `acquirelocks(l, threadId)`. Lembramos que este método é bloqueante, ou seja, só retorna quando for garantido que não existem mais operações a decorrer, relativamente aquele ficheiro.

```
1 public Resource writeOperation(String path){
2 
3   Lock l = new WriteLock(path, order, threadId);
4   ContentionManagerImpl.acquirelocks(l);
5 
6   doOperation;
7 
8   ContentionManagerImpl.releaseLocks(l);
9 }
```

Figura 4.14: Exemplo de um método onde é utilizado o `ContentionManager`.

Ao contrário das operações de escrita, as operações de leitura podem executar em concorrência com outras operações de leitura. Assim, quando no método `acquirelocks` é passado como argumento um objecto `ReadLock`, este só ficará bloqueado no caso de já haver uma operação de escrita em execução ou bloqueada à espera da sua vez.

A nossa solução centra-se sobretudo na classe `ContentionManagerImpl`, que implementa a interface apresentada na secção 3.2.3. Esta classe, que é estática, gere todos os acessos aos recursos, definindo a ordem como estes são feitos, e é composta por uma tabela onde a cada recurso é associada uma fila de *locks*.

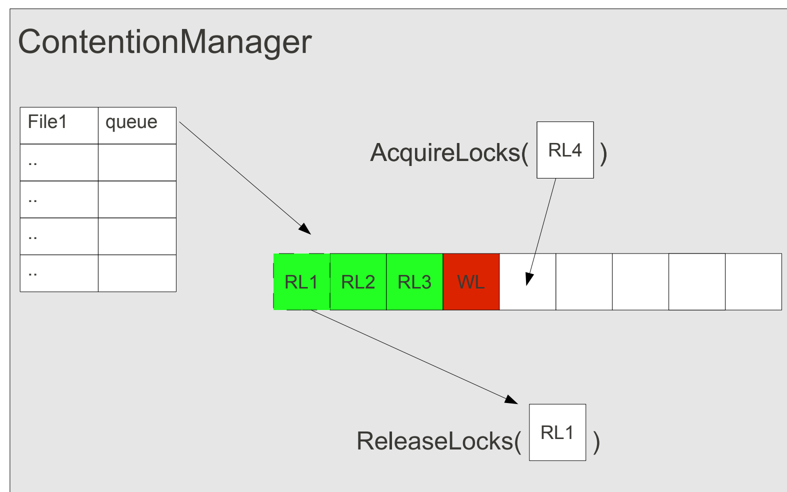

Figura 4.15: Exemplificação da forma como o mecanismo de contenção funciona.

Quando é chamado o método `acquireLocks(l)`, o objecto que é passado como argumento do tipo `Lock` é inserido na fila respectiva ao ficheiro. Caso esta ainda não exista, é criada uma entrada nova na tabela, e só será removido no fim da operação ser executada, ou seja, quando for chamado o método `releaseLocks(l)`. No caso de o *lock* ser do tipo `ReadLock`, apenas terá de esperar se já houver um *lock* de escrita mais antigo. No exemplo da Figura 4.15, o *lock* RL4 ficará bloqueado até que o *lock* WL1, de escrita, termine a sua execução.

Em termos práticos, existe uma variável booleana que se encontra a `true` sempre que exista uma operação de escrita em espera ou em execução. Quando entra no sistema outra operação, e esta variável tiver o valor `true`, é chamado o método `wait()` no *lock*, que leva a que a operação fique bloqueada até que seja feito um `notify()`.

Quando terminar a execução da operação de escrita e for chamado o método `releaseLocks()` será percorrida a fila de *locks* até que seja encontrado outro *lock* de escrita ou que não existam mais elementos, sendo chamado o método `notify()` em todos os elementos intermédios de leitura, de modo a que possam executar concorrentemente. Nesta situação, um *lock* de escrita só passará a executar se não existirem *locks* intermédios de leitura, caso contrário, só será notificado quando a última operação de leitura terminar.

No âmbito do JNFS, onde foi testado este trabalho, este mecanismo foi utilizado da forma como se pode observar na Figura 4.16 onde se encontra o exemplo da operação `read` do sistema de ficheiros que necessita de um *lock* de leitura, `rl`. Este é criado com base no caminho do ficheiro onde se vai efectuar a operação e no identificador da *thread* que está a executar o método. Quando é invocado o método `acquireLocks` é passada uma lista de *locks*, que neste caso apenas contém um elemento, e a ordem da operação.

A ordem é obtida através do método `getOrder`. Uma vez que na aplicação não temos acesso ao número de ordem da operação é necessário obter essa informação. Assim, ao obter uma nova operação vinda da biblioteca e antes que esta seja entregue à aplicação dentro da classe `ServerSocket` a ordem da operação, associada ao endereço do cliente,

```
1
2  public Block read(int fd) throws java.io.IOException
3    {
4      int order = getOrder(UnicastRemoteObject.getClientHost());
5      List<Lock> lockList = new ArrayList<Lock>();
6      ReadLock rl = new ReadLock(table.getPath(fd), Thread.currentThread().getId());
7      lockList.add(rl);
8      ContentionManagerImpl.acquireLocks(lockList, order);
9        java.io.FileInputStream fis = getRMIFileInputStream(fd);
10       Block block = new Block();
11       block.length = fis.read(block.buf, 0, Block.BUFSIZE);
12     ContentionManagerImpl.releaseLocks(rl);
13     return block;
14   }
```

Figura 4.16: Exemplo de um método onde é utilizado o `ContentionManager` no âmbito do JNFS.

é colocada num mapa partilhado com a aplicação. No método `getOrder` vamos buscar a este mapa o número de ordem da última operação correspondente ao cliente que efectuou a invocação, ou seja, da invocação corrente. Note-se ainda que, o endereço do cliente é obtido através do método `UnicastRemoteObject.getClientHost()`.

Validação Experimental

Neste capítulo iremos apresentar a metodologia utilizada para validar o nosso trabalho, tal como os resultados experimentais obtidos. Em primeiro lugar será feita uma comparação entre as nossas duas aproximações, em que as comunicações são feitas através de UDP e de um algoritmo de tolerância a falhas bizantinas, em relação à implementação actual do RMI. Numa segunda fase serão apresentados dados que revelam o impacto que a execução especulativa pode ter neste tipo de sistemas.

5.1 Metodologia

De modo a avaliar as implicações, em termos de performance, das nossas implementações da camada de transporte do RMI do Java, implementámos um *benchmark* de teste baseado no JNFS, já apresentado anteriormente, composto por 5 fases. Este teste visa, sobretudo, comparar a latência das operações. Numa última fase avaliámos também a forma como o uso de execução especulativa pode melhorar a eficiência destes sistemas.

Os testes relativos à versão original do RMI e da nossa versão baseada em UDP foram feitos entre duas máquinas, um cliente e um servidor, enquanto que os testes à nossa versão do RMI com tolerância a falhas bizantinas foram feitos assumindo uma réplica com comportamento arbitrário, daí existirem 4 servidores. Foi ainda avaliado o impacto da execução de operações concorrentes nos servidores, por comparação a um método de execução sequencial.

Cada teste foi corrido 5 vezes, tendo sido eliminado o melhor e o pior resultado de modo a calcular a média dos três restantes.

O ambiente de testes é constituído por um *cluster* de 5 nós, com servidores Dell, com processadores AMD Opteron Quad-core 2376 a 2.3Ghz, 16 Gb de memória, a executar o

sistema operativo Linux Debian 5. As máquinas estão conectadas por uma rede Ethernet 1Gb.

5.2 Benchmark

Na secção 3.3 apresentámos o JNFS [Rad97], um sistema de ficheiros distribuído implementado recorrendo ao RMI para efectuar as comunicações entre cliente e servidor. Nesta secção iremos apresentar um *benchmark* implementado por nós com o objectivo de testar o sistema e comparar as nossas implementações com o RMI original em termos performance.

O *benchmark*, descrito em [dSV10], é composto por 5 fases, independentes entre elas, e a sua performance é avaliada contabilizando o tempo que cada uma delas demora a executar, tal como o tempo total. O *benchmark* é então composto pelas seguintes operações:

- **Fase 1** – Esta fase consiste na criação e remoção de 6 directorias de forma recursiva.
- **Fase 2** – Cópia, e posterior remoção, de 7 ficheiros com tamanho máximo de 4kb.
- **Fase 3** – Verificação de algumas propriedades, como o tamanho e as permissões, de 6 ficheiros, de forma sequencial.
- **Fase 4** – Leitura de um ficheiro de 4kb *byte* a *byte*.
- **Fase 5** – Criação de um ficheiro com 4kb.

Na Figura 5.1 é apresentado o código da implementação da segunda fase do *benchmark*. Esta é composta por um ciclo principal que cada iteração contém duas operações: (1) copia de um ficheiro entre duas directorias; (2) remoção do ficheiro copiado. Cada uma destas operações acaba por ter duas fases distintas, devido à forma como o JNFS funciona. Em cada operação é sempre necessário em primeiro lugar obter uma referência para um, ou mais, objectos que serão responsáveis pela execução da operação. Na segunda fase a operação é executada. Por exemplo, para remover um ficheiro é necessário obter primeiro uma referência para um objecto do tipo `RemoteFile`, que representa o ficheiro, e depois executar o método `delete()`.

5.2.1 Resultados

Os nossos testes permitem avaliar de que forma as modificações introduzidas no RMI afectam a latência do JNFS. Este impacto foi avaliado de duas formas distintas: (1) através do tempo total do *benchmark* e de cada uma das suas fases; (2) através do número total de operações efectuadas.

```
// Copy / remove a source tree containing 7 files up to 4 Kb
public void phase2(){
  try {
    for(int i = 0; i< Util.phase2Files.length; i++){
      // Copy a file
      RemoteFileInputStream is = server.getFileInputStream(
            Util.srcDir + Util.phase2Files[i]);
      RemoteFileOutputStream os = server.getFileOutputStream(
            Util.destDir + Util.phase2Files[i]);
      byte b[] = new byte[10000];
      int count;
      while((count = is.read(b)) > 0) {
        os.write(b, 0, count);
      }
      os.close();
      is.close();
      //  Remove a file
      String dir = Util.destDir + Util.phase2Files[i];
      RemoteFile rf = server.getFile(dir);
      System.out.println("Deleting␣file␣<" + dir + ">␣:␣" + rf.delete());
    }
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

Figura 5.1: Fase 2 do *benchmark*.

5.2.1.1 Tempo total

Na Figura 5.2 encontramos a comparação entre os tempos gastos por cada fase do *benchmark* em cada uma das três configurações (TCP, UDP e BFT), tal como o tempo total. Note-se que estes tempos correspondem à execução de um único cliente e um único servidor.

A primeira conclusão que podemos tirar é que entre a nossa primeira versão, baseada em UDP, acaba por levar a uma pouco significativa perda de performance, na ordem dos 8%, enquanto que a segunda versão aumenta bastante a latência do sistema relativamente à implementação actual do RMI. Este aumento da latência é natural, uma vez que as operações antes de serem executas têm de ser ordenadas pelas réplicas através de 3 rondas de comunicação entre elas.

A segunda conclusão é que a segunda fase do *benchmark* ocupa uma grande parte do tempo total, cerca de 70%, em todas as versões. Como tal, é um possível alvo de melhoramentos onde poderemos ter um ganho significativo em termos de performance. A par desta, as fases 1 e 3 ocupam uma razoável percentagem do tempo total de execução. Esta característica é transversal às três implementações, o que leva a que seja nestas três fases que seja estudado o uso de execução especulativa.

5.2.1.2 Número de operações

Outra métrica importante neste tipo de sistemas é o número de operações efectuadas por segundo. Na Figura 5.3 podemos ver a comparação entre o número de operações por

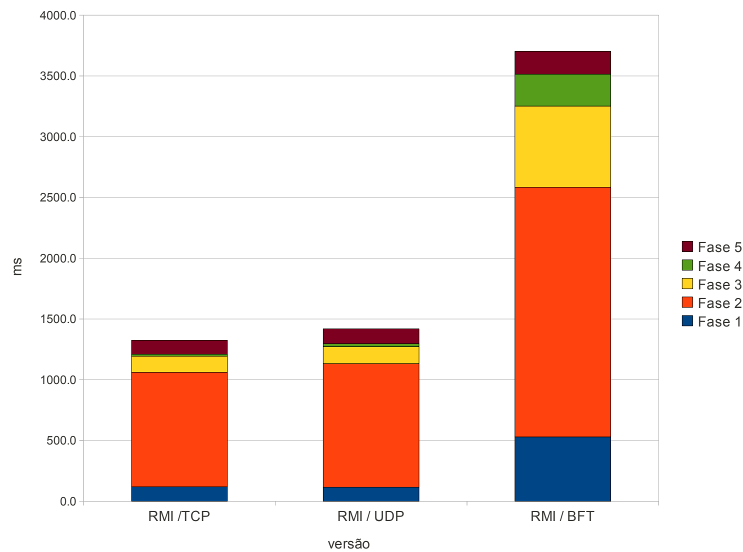

Figura 5.2: Comparação dos tempos obtidos nas três versões.

segundo obtidos em cada uma das três versões durante a execução do *benchmark*, com 1, 2, 4 e 8 clientes a executar em simultâneo.

Podemos ver que o número de operações cresce à medida que aumentamos o número de clientes em execução. As versões baseadas em TCP e UDP têm um crescimento bastante mais acentuado relativamente à versão com tolerância a falhas bizantinas. Este facto é normal, uma vez que nesta última versão, há muito tempo gasto em comunicações na ordenação das mensagens.

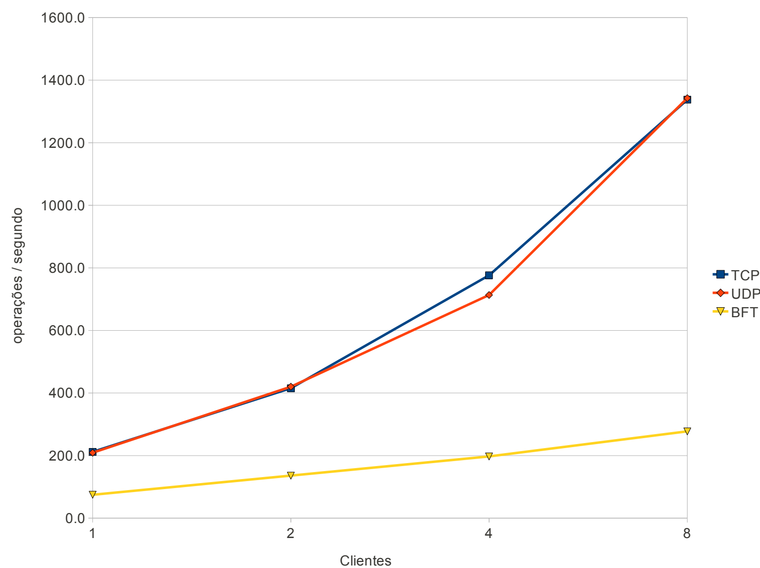

Figura 5.3: Comparação das operações por segundo obtidas nas três versões.

5.3 Especulação entre operações independentes

De forma a melhorar a performance do sistema e diminuir algum do custo inerente ao uso de algoritmos de tolerâncias a falhas bizantinas, decidimos introduzir especulação no cliente. Nesta primeira fase introduzimos especulação baseada em futuros naquelas fases que ocupavam uma maior parte do tempo total da execução do benchmark.

Como explicámos na Secção 5.2, para efectuar uma operação no JNFS é sempre necessário obter uma referência para um objecto remoto, a partir do qual será invocado o método respectivo à operação. Esta última depende da primeira, o que leva a que não possam ser executadas concorrentemente. Nesta fase procurámos identificar as relações de dependência entre as operações, procurando identificar operações que, se executadas em paralelo, permitiriam obter um ganho de performance substancial.

Na Figura 5.4 podemos observar a ordem como as operações são efectuadas na fase 2 do *benchmark* ao ser utilizada especulação em algumas operações. Por exemplo, as invocações dos métodos `srv.getInputStream(f)` e `srv.getOutputStream(f)` podem ser executadas em paralelo, recorrendo a futuros para executar uma delas. Desta forma, iremos poupar o tempo de execução da operação mais curta.

srv.getInputStream(f)	srv.getOutputStream(f)
is.read(b)	
os.write(b, 0, count)	
is.close()	os.close()
srv.getFile(f)	
file.delete()	

Figura 5.4: Operações efectuadas na fase 2 do *benchmark*, com especulação.

5.3.1 Resultados

5.3.1.1 Tempo total

Na Figura 5.5 podemos encontrar a comparação entre os tempos obtidos para cada fase, após a introdução de especulação nas fases 1, 2 e 3.

Na primeira fase do *benchmark*, apenas na versão original do RMI não se verificaram melhorias, enquanto que na versão baseada em UDP houve uma redução de 17% do tempo e na versão baseada em BFT de 17.5%. Na segunda fase, que ocupava uma parcela significativa do tempo, registámos um ganho de 15.9% na versão original, enquanto que nas versões baseadas em UDP e BFT alcançámos um ganho de 17.5% e 24.6% respectivamente. Já na terceira fase observámos resultados um pouco contraditórios, uma vez que na versão BFT até se regista uma perda de 17.4%, enquanto que nas restantes registámos ganhos de 19.8%, na versão baseada em UDP, e de 21.7%, na versão original.

No tempo total do *benchmark*, registámos uma melhoria de 13% na versão original do RMI, 15.7% na versão baseada em UDP e 12.6% na versão BFT.

Podemos então constatar que, na grande parte dos casos, a duração das fases foi reduzida de uma forma substancial, demonstrando os benefícios do uso de execução especulativa neste tipo de sistemas.

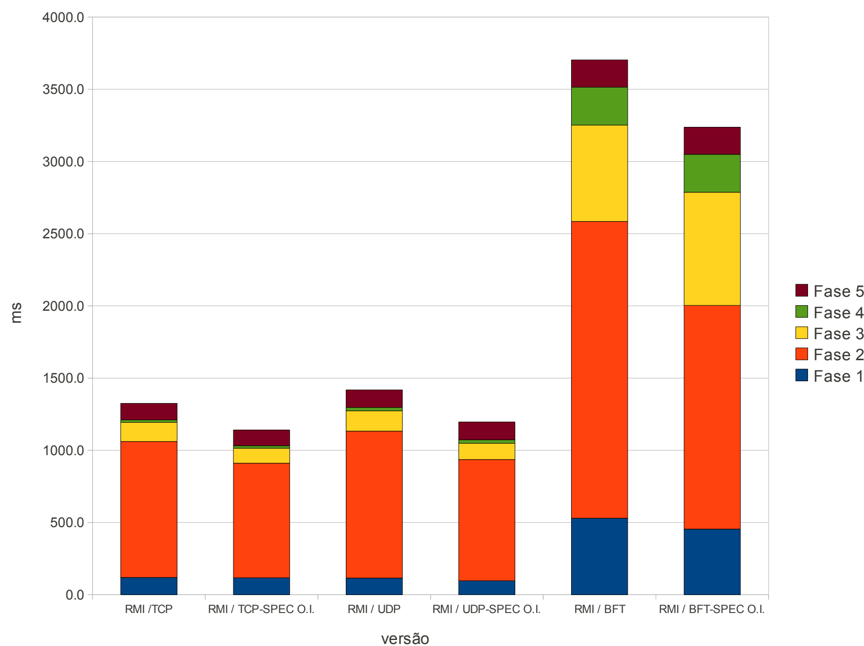

Figura 5.5: Comparação dos tempos obtidos nas três versões.

5.3.1.2 Número de operações

Na Figura 5.6 podemos novamente observar que o número de operações cresce de forma bastante mais acentuada nas versões sem replicação, relativamente à versão BFT, à medida que aumentamos o número de clientes.

Relativamente à versão anterior, sem especulação, nas versões baseadas em TCP e UDP os valores apenas apresentam melhorias quando se encontra um cliente em execução, sendo esta de 8% em ambos os casos. Já na versão replicada, existe sempre melhoria, verificando-se um aumento no número de operações de 10% com 1 e 4 clientes, enquanto que com 2 e 8 clientes, nota-se uma melhoria de de apenas 5%.

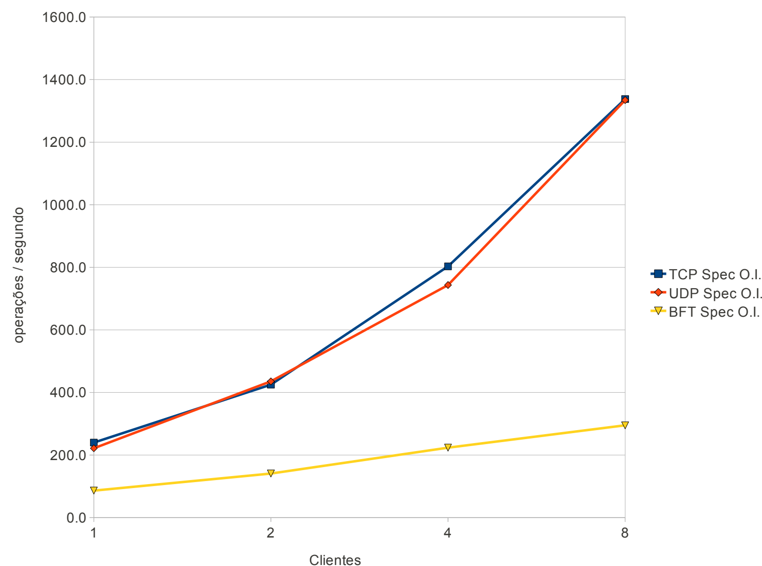

Figura 5.6: Comparação das operações por segundo obtidas nas três versões.

5.4 Especulação entre operações dependentes

Apesar dos resultados obtidos na secção anterior, existe ainda espaço para melhoramentos, em especial na fase 2 do *benchmark*. Recordamos que esta fase é composta por um ciclo que executa várias operações a cada iteração.

Como explicámos, muitas operações dependem dos resultados de operações anteriores para poderem executar. Assim, podemos dividir uma iteração em 3 fases distintas em que cada uma depende dos resultados da anterior. Uma forma de podermos aumentar o paralelismo neste tipo ciclos é criar um *pipeline* de operações.

Assim, como podemos ver na Figura 5.7, em cada iteração do ciclo cada fase encontra-se num ponto diferente. Enquanto que na primeira iteração apenas a primeira fase é executada, na segunda também será executada a segunda fase, com base nos resultados obtidos na iteração anterior, e na iteração seguinte, será executada a terceira fase, também baseada nos resultados obtidos anteriormente. Assim, podemos executar as 3 fases em

paralelo, poupando o tempo de espera entre elas.

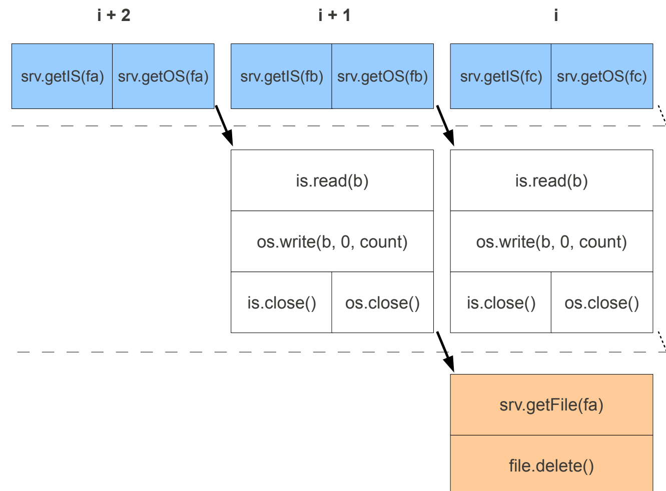

Figura 5.7: Operações efectuadas na fase 2 do *benchmark*, com especulação.

5.4.1 Resultados

5.4.1.1 Tempo total

Na Figura 5.8 podemos encontrar a comparação entre os tempos obtidos para cada fase, após a utilização desta última técnica de especulação na fase 2 do *benchmark*.

Podemos observar que houve uma redução significativa no tempo de execução nesta fase. Na versão original do RMI os ganhos foram de 33%, na versão baseada em UDP 36.7% e na versão BFT de 45.4%. Já o tempo total de execução do *benchmark* também sofreu uma redução drástica, verificando-se um corte de 25.4% na versão sem modificações do RMI, 29.4% na versão baseada em UDP e 24.8% na versão BFT.

5.4.1.2 Número de operações

Na Figura 5.9 encontramos um gráfico com a comparação do número de operações efectuadas pelas várias versões com esta nova forma de especulação.

Observa-se novamente um crescimento acentuado do número de operações à medida que aumentamos o número de clientes. No entanto, nesta versão de especulação, os ganhos são mais relevantes em todas implementações do RMI.

Enquanto que na execução com 1 e 2 clientes os ganhos são bastante semelhantes nas 3 versões, apresentando ganhos de cerca de 25% para um cliente em execução e 11% para dois, quando comparados com o modelo de execução sequencial, com o aumento do número de clientes em execução, é nas versões BFT e UDP que se registam os maiores ganhos, com um aumento de 18% e 16% respectivamente quando existem 4 clientes a executar e 15% e 10% no caso de existirem 8 clientes. Já na versão TCP, os ganhos são menos significativos, de 9%.

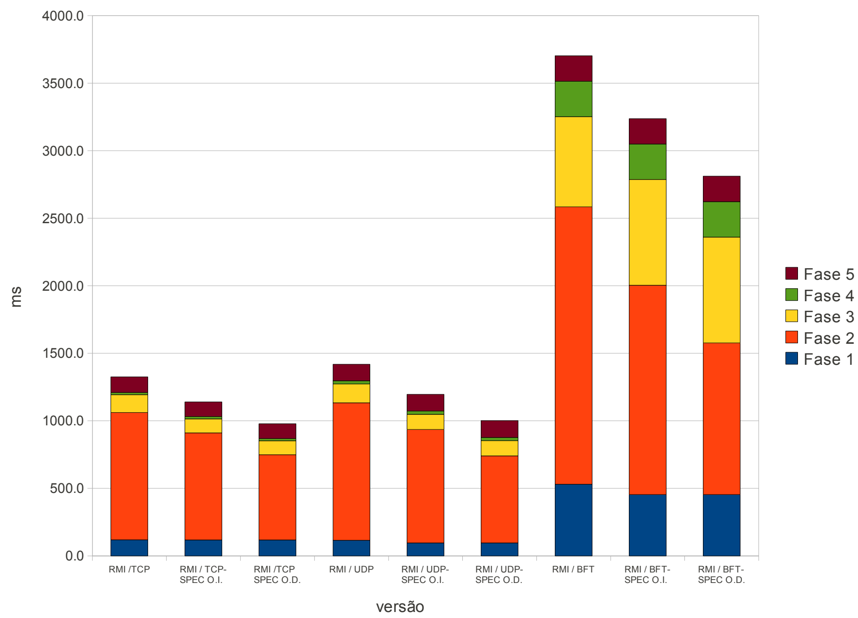

Figura 5.8: Comparação dos tempos obtidos nas três versões.

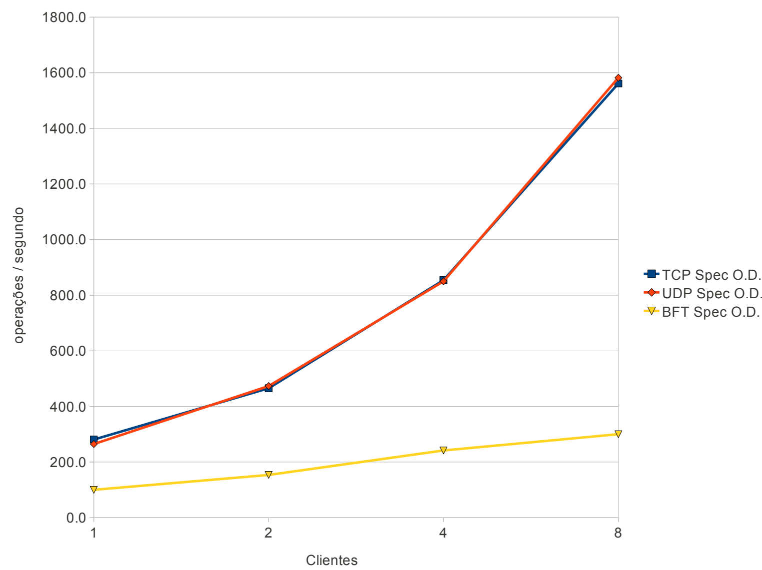

Figura 5.9: Comparação das operações por segundo obtidas nas três versões.

5.5 Comparação

De uma forma geral, podemos afirmar que a execução especulativa permite alcançar consideráveis melhorias de performance.

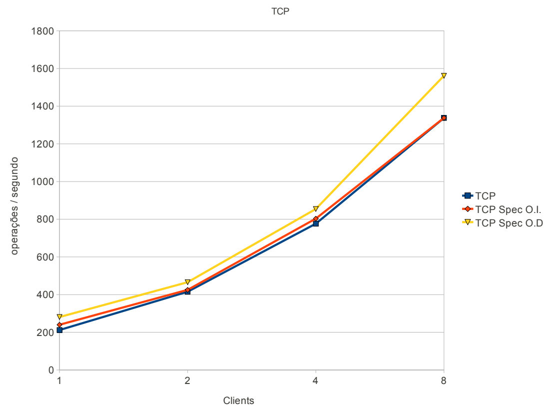

Figura 5.10: Comparação do número de operações por segundo utilizando TCP.

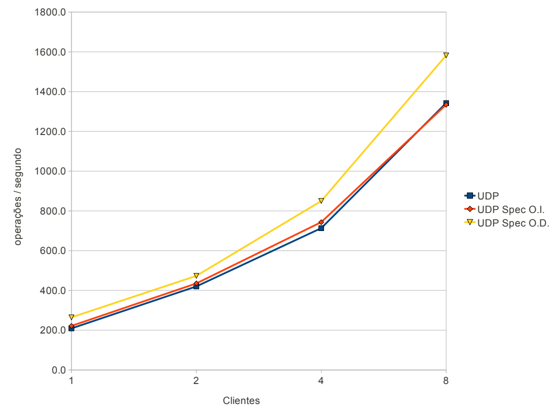

Figura 5.11: Comparação do número de operações por segundo utilizando UDP.

Nas versões TCP e UDP, como podemos constatar nas Figuras 5.10 e 5.11, o número de operações por segundo é sempre mais elevado quando é utilizada especulação baseada em futuros no cliente. Podemos observar que ambas as versões de especulação são mais eficientes com 1, 2, 4 e 8 clientes. Em especial, a execução especulativa entre operações dependentes leva a um aumento significativo do número de operações por segundo em todos os casos, sendo que com 8 clientes a executar, obtivemos ganhos de 14% na versão TCP e 15% na versão UDP.

Na Figura 5.12 podemos observar que existem sempre ganhos ao introduzir execução especulativa no nosso sistema, utilizando a versão com tolerância a falhas bizantinas. No caso da especulação entre operações independentes verifica-se um ganho de 11% e 18% caso seja utilizada especulação entre operações dependentes, para 4 clientes em execução. Já no caso de se encontrarem 8 clientes a executar, os ganhos são de 6% e 10% respectivamente.

5.6 Execução Concorrente no Servidor

Quando descrevemos a biblioteca de comunicação utilizada no nosso trabalho referimos que esta apresenta um modo de execução das operações nos servidores concorrente, ao invés do que é comum neste tipo de sistemas, onde as operações são executadas sequencialmente. Este facto permite que seja melhorado o desempenho dos servidores, beneficiando das arquitecturas compostas por vários processadores, diminuindo assim o tempo de espera para o cliente.

Nesta secção iremos avaliar o impacto desta execução concorrente relativamente a

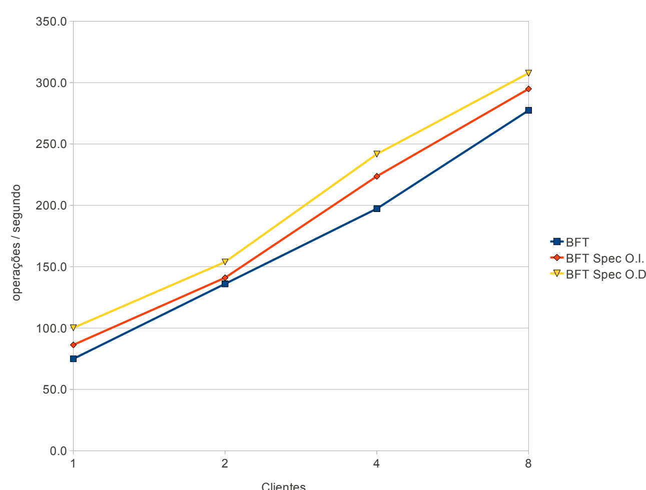

Figura 5.12: Comparação do número de operações por segundo utilizando BFT.

uma versão da mesma biblioteca que executa as operações sequencialmente, ou seja, pretendemos avaliar o impacto de executar várias operações em simultâneo nos servidores, ao invés de apenas uma de cada vez. Este impacto foi avaliado através da comparação entre o número de operações por segundo alcançado por estas duas versões da biblioteca na execução do *benchmark* utilizado nos testes anteriores. A metodologia utilizada foi semelhante aos testes anteriores e foram utilizados 1, 2, 4 e 8 clientes.

Na Figura 5.13 podemos observar os resultados obtidos depois de correr o *benchmark* sem qualquer tipo de especulação. É possível constatar que com apenas um cliente não existem diferenças assinaláveis, sendo o número de operações por segundo bastante semelhante. Este facto é natural, uma vez que as operações são, por natureza, sequenciais não existindo paralelismo. À medida que aumentamos o número de clientes já é possível constatar que existe uma melhoria na performance da versão concorrente em relação à versão sequencial. Esta melhoria é mais acentuada com 8 clientes, situando-se na ordem dos 13%, o que se justifica pelo facto de existirem mais operações em execução em simultâneo, e provavelmente fosse mais notória se as operações levassem mais tempo a executar nas réplicas.

As versões do *benchmark* com execução especulativa apresentam uma diferença mais acentuada entre as duas bibliotecas, como podemos constatar nas Figuras 5.14 e 5.15. Mais uma vez é possível verificar que executando apenas um cliente não existe uma grande diferença entre as duas bibliotecas, uma vez que as operações demoram pouco tempo a executar no servidor. No entanto, à medida que é aumentado o número de clientes também existe um maior ganho de performance. Com 4 clientes a executar este ganho é de 26% utilizando especulação entre operações independentes e 21% caso seja utilizada especulação entre operações dependentes. Já utilizando 8 clientes, os ganhos são de 16% e 18% respectivamente.

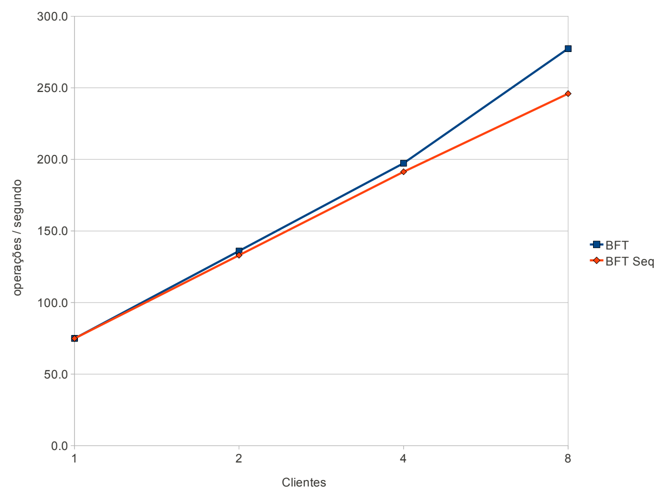

Figura 5.13: Comparação do número de operações por segundo entre as duas bibliotecas sem especulação.

Assim, podemos afirmar que o facto de a biblioteca de comunicação permitir a execução concorrente de operações pode trazer benefícios à performance global do sistema, benefícios esses que seriam mais acentuados no caso no caso de as operações demorarem mais tempo a executar no servidor.

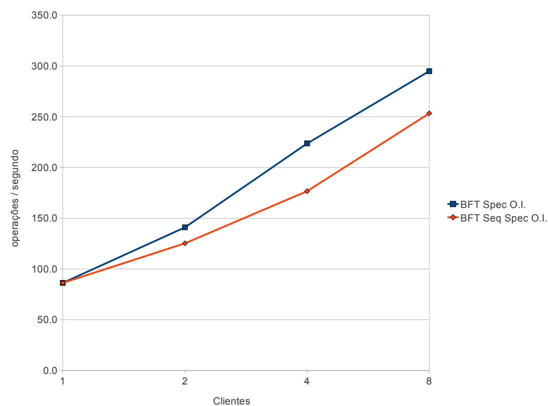

Figura 5.14: Comparação do número de operações por segundo entre as duas bibliotecas utilizando especulação entre operações independentes.

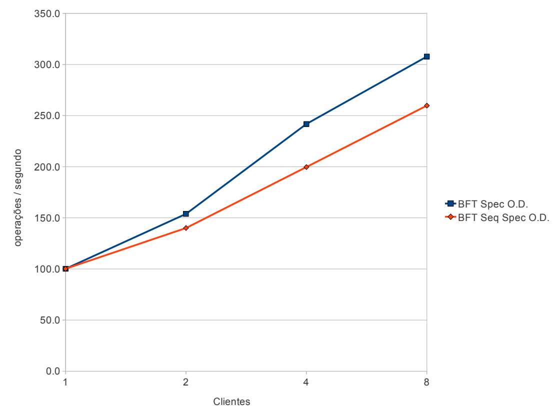

Figura 5.15: Comparação do número de operações por segundo entre as duas bibliotecas utilizando especulação entre operações dependentes.

6

Conclusão

O RMI apresenta um modelo de programação de aplicações distribuídas que procura assemelhar-se, tanto quanto possível, ao modelo de programação de aplicações normais. A sua principal vantagem é a forma como permite que o programador se abstraia dos detalhes das comunicações entre objectos.

No entanto, sendo baseado numa arquitectura cliente/servidor, não é possível criar aplicações robustas, isto é, fiáveis, sem definir de forma explícita a arquitectura do sistema.

Um mecanismo de replicação implícito, como o nosso, permite suprimir essa lacuna, não implicando alterações de maior na implementação tanto do cliente, como do servidor.

Não é trivial implementar este tipo de técnicas, uma vez que o RMI não está preparado para comunicações do tipo 'um para muitos'. No caso de as aplicações se basearem em máquinas de estados replicadas é particularmente complexo, uma vez que o RMI possui fontes de não-determinismo nas operações, como explicámos na secção 4.3. Apesar disso, o facto de o RMI se encontrar estruturado em camadas facilita a alteração da camada de comunicação.

A utilização de algoritmos de tolerância a falhas prejudica de uma forma considerável a performance dos sistemas, quando comparado com a implementação de raiz do RMI, principalmente devido ao elevado número de mensagens trocadas entre as réplicas.

Este impacto negativo foi minimizado através do uso de execução especulativa no cliente.

Através dos nossos resultados de validação, podemos afirmar que este impacto pode ser minimizado até um ponto em que a sua utilização se torna aceitável.

6.1 Trabalho Futuro

Existem algumas melhorias que poderão ser introduzidas no nosso sistema no futuro.

6.1.1 Comparação de Resultados

Como explicámos no documento, os resultados das invocações remotas têm de ser comparados, de modo a ser considerados correctos por parte do cliente. O RMI coloca em algumas mensagens informação acerca do servidor, como o seu endereço ou o seu *ObjId*, o que faz com que as mensagens dos vários servidores sejam diferentes e o cliente não as considere válidas. No nosso trabalho, este problema é ultrapassado trocando nas mensagens directamente os identificadores dos objectos das réplicas secundárias pelo identificador da réplica primária. Uma forma de tornar este processo mais homogéneo seria avaliar apenas o excerto da mensagem que corresponde ao resultado da operação.

6.1.2 Modelo de Execução Especulativa

O nosso modelo de especulação é apenas baseado em futuros, não sendo feita qualquer previsão de resultados. Seria interessante, como forma de melhorar o desempenho das aplicações, implementar um modelo de especulação mais complexo, baseado na previsão de resultados.

6.1.3 Aplicação a outros modelos de falhas

O objectivo deste trabalho foi fornecer tolerância a falhas bizantinas em aplicações RMI. No entanto, existem outras falhas, como as falhas *fail-stop*, que também são prejudiciais para estas aplicações. Seria, assim, interessante estudar formas de implementar outros mecanismos de tolerância a falhas seguindo o mesmo modelo.

6.1.4 Utilização de memória transaccional

A memória transaccional é um mecanismo mecanismo de controlo de concorrência cada vez mais popular entre a comunidade devido à sua maior facilidade de utilização em relação aos mecanismos de *locks*. Apesar de ainda ser uma tecnologia em fase de amadurecimento, seria interessante estudar a sua utilização como forma de controlar a concorrência nos servidores do nosso sistema.

6.1.5 Mecanismo de Locks automático

A utilização do mecanismo de gestão de recursos implementado implica que o programador altere os métodos explicitamente. Seria assim interessante estudar formas de tornar este mecanismo automático, facilitando a sua utilização.

Bibliografia

[CDK05] George Coulouris, Jean Dollimore, e Tim Kindberg. *Distributed Systems Concepts and Design*. Addison-Wesley, 2005. ISBN: 0-321-26354-5.

[CG99] Fay Chang e Garth A. Gibson. Automatic i/o hint generation through speculative execution. In *Proceedings of the third symposium on Operating systems design and implementation*, OSDI '99, pág. 1–14, Berkeley, CA, USA, 1999. USENIX Association.

[CL02] Miguel Castro e Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, Nov 2002.

[CLM+08] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, e Andrew Warfield. Remus: high availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pág. 161–174, Berkeley, CA, USA, 2008. USENIX Association.

[Cou] Emanuel Couto. Speculative execution by using software transactional memory. Tese de Mestrado, Faculdade de Ciências e Tecnologia - Universidade Nova de Lisboa.

[dSV10] Giuliana Teixeira dos Santos Veronese. *Intrusion Tolerance in Large ScaleNetworks*. Tese de Doutoramento, Universidade de Lisboa, 2010.

[KAD+07] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, e E. Wong. Zyzzyva: speculative byzantine fault tolerance. In *Symposium on Operating Systems Principles (SOSP)*, Oct 2007.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, July 1978.

[Lam82] L. Lamport. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4:382–401, 1982.

[LDZN03] Xiao-Feng Li, Zhao-Hui Du, Qing-Yu Zhao, e Tin-Fook Ngai. Software value prediction for speculative parallel threaded computations. In *Proceedings of the First Value-Prediction Workshop*, pág. 18–25, San Diego, CA, June 2003.

[LEG10] Shaoshan Liu, Christine Eisenbeis, e Jean-Luc Gaudiot. Value Prediction and Speculative Execution on GPU. *International Journal of Parallel Programming*, pág. 1–20, November 2010.

[LK06] Michael E. Locasto e Angelos D. Keromytis. Speculative execution as an operating system service. Relatório técnico, Department of Computer Science, Columbia University, May 2006.

[MG99] Pedro Marcuello e Antonio González. Clustered speculative multithreaded processors. In *Proceedings of the 13th international conference on Supercomputing*, ICS '99, pág. 365–372, New York, NY, USA, 1999. ACM.

[NCF06] Edmund B. Nightingale, Peter M. Chen, e Jason Flinn. Speculative execution in a distributed file system. *ACM Trans. Comput. Syst*, pág. 361–392, 2006.

[NMMS00] N. Narasimhan, L. E. Moser, e P. M. Melliar-Smith. Transparent consistent replication of java rmi objects. In *Proceedings of the International Symposium on Distributed Objects and Applications*, pág. 17–, Washington, DC, USA, 2000. IEEE Computer Society.

[OL02] J. Oplinger e M.S. Lam. Enhancing software reliability using speculative threads. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*, Oct 2002.

[PRHL08] Nuno M. Preguiça, Rodrigo Rodrigues, Cristóvão Honorato, e João Lourenço. Byzantium: Byzantine-fault-tolerant database replication providing snapshot isolation. In *HotDep*, 2008.

[Rad97] Michael John Radwin. A java network file system for network computers, 1997.

[RMI] Rmi transport protocol. `http://download.oracle.com/javase/1.3/docs/guide/rmi/spec/rmi-protocol3.html`.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22:299–319, December 1990.

[SCZM00] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, e Todd C. Mowry. A scalable approach to thread-level speculation. *SIGARCH Comput. Archit. News*, 28:1–12, May 2000.

[WCN+09] Benjamin Wester, James Cowling, Edmund B. Nightingale, Peter M. Chen, Jason Flinn, e Barbara Liskov. Tolerating latency in replicated state machines through client speculation. In *Proceedings of the Sixth Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, Massachusetts, Abril 2009.