UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

FTRMI: PLATAFORMA TRANSPARENTE TOLERANTE A FALTAS PARA INVOCAÇÕES REMOTAS

Diogo André Mota dos Reis

DISSERTAÇÃO

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

2012

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

FTRMI: PLATAFORMA TRANSPARENTE TOLERANTE A FALTAS PARA INVOCAÇÕES REMOTAS

Diogo André Mota dos Reis

DISSERTAÇÃO

Orientada pelo Prof. Doutor Hugo Alexandre Tavares Miranda

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

2012

Resumo

As Chamadas a Procedimentos Remotos (RPC) têm como objectivo facilitar a comunicação entre processos, mascarando-a com uma sintaxe próxima da invocação a procedimentos mas ocultando os detalhes de comunicação. Contudo, devido à evolução dos paradigmas de programação, foi necessário encontrar uma solução para a programação Orientada a Objectos (OO). As Plataformas de Objectos Distribuídos (DOF) disponibilizam as mesmas características de um RPC adaptando a tecnologia a este paradigma. A Invocação Remota de Métodos (RMI) cumpre os objectivos de um DOF. Ainda assim, a especificação desta tecnologia para Java (JRMI) é totalmente dependente do modelo cliente/servidor criando um ponto de falha no lado do servidor.

As aplicações distribuídas devem apresentar uma qualidade de serviço, nomeadamente, tolerância a faltas e escalabilidade que satisfaça os utilizadores. Uma possibilidade para os sistemas computacionais cumprirem estes requisitos é a distribuição do serviço por vários servidores distintos, incentivando a tolerância a faltas e distribuição de carga. Contudo, uma sistema distribuído é mais complexo que um sistema centralizado, devido à maior diversidade de problemas a resolver.

Uma abordagem clássica de tolerância a faltas é a replicação activa, onde todas as réplicas mantêm um estado coerente por executarem apenas operações deterministas e sempre pela mesma ordem. Recorrendo ao conceito de máquina de estados distribuída que concretiza a replicação activa, é possível a criação de aplicações tolerantes a faltas de forma transparente para os servidores.

Este trabalho apresenta a plataforma Fault-Tolerante Remote Method Invocation (FTRMI), que proporciona ao JRMI a capacidade de replicação activa de objectos remotos. A plataforma é disponibilizada sob a forma de camada de código intermédio tornando-se totalmente transparente para o cliente e não sendo necessário qualquer alteração de código no lado servidor. O grande objectivo desta plataforma é manter transparência total para aplicações existentes implementadas em JRMI. O FTRMI foi comparado com uma solução não transparente, mas que fornece uma qualidade de serviço semelhante.

Palavras-chave: Invocação remota, Camada de Código Intermédio, JRMI, Tolerância a faltas

Abstract

In computer science, a Remote Procedure Call (RPC) is a communication mechanism that aims to hide the communication details from the programmer. Due to the evolution of programming paradigms, it was deemed necessary to apply this technique to Object Oriented (OO) programming. The solution was found on Distributed Object Frameworks (DOF), which offer the same benefits of the RPC technology, but for this paradigm. Java Remote Method Invocation (JRMI) meets the requirements of a DOF, but, by using a client/server communication model, suffers from a potential single point of failure on the server side.

The distributed applications require higher quality of service and, in particular, fault tolerance and scalability. Computational systems are able to fulfil these requirements by employing multiple machines, encouraging fault tolerance and load distribution. A distributed system is, by necessity, more complex than a centralized one, to deal with problems such as heterogeneity and synchronism. However, the Group Communication Systems (GCS), whose main objective is to provide a simple interface that implements several concepts, can facilitate the implementation of replication and fault tolerance.

A classical approach to achieve fault tolerance is active replication, wherein all replicas maintain a consistent state by executing deterministic operations in the same order. Using the concept of distributed state machine that implements active replication, it is possible to create fault tolerant applications transparently to the servers.

This work presents Fault-Tolerant Remote Method Invocation (FTRMI) framework, that enables JRMI to support active replication on the remote objects. The platform is available in the form of middleware that becomes totally transparent to the client and doesn't require any code changes on the server. The main objective of this platform is to maintain total transparency to existing applications already using JRMI. The FTRMI was compared to a different solution that provides a similar quality of service, but isn't transparent.

Keywords: Remote Invocation, Middleware, JRMI, Fault Tolerance

Índice

Lista de Figuras

Capítulo 1

Introdução

1.1 Motivação

A Comunicação entre Processos (IPC) [9] tem como objectivo permitir a troca de informação entre dois processos. Esta troca de informação também pode ser efectuada através de uma rede de computadores, ou qualquer outro meio de comunicação. Os diferentes tipos de IPC são, Chamada a Procedimentos Remotos (RPC) [23], Memória Partilhada, Sincronização e Troca de Mensagens. Todos estes mecanismos possuem diferentes características e consequentemente diferentes objectivos, dos quais podem ser, partilha de informação, modularidade, separação de privilégios ou paralelização. No âmbito deste projecto vamos focar o conceito do RPC.

Um RPC permite a execução de um procedimento num outro processo, com outro espaço de endereçamento, mas mantendo a semântica de uma execução local. O aparecimento de linguagens orientadas a objectos tornou o conceito de RPC obsoleto. Desde então, este conceito foi expandido para facilitar a sua integração em linguagens de programação orientadas a objectos como Plataformas de Objectos Distribuídos (DOF) [35]. Algumas implementações destas plataformas são o Distributed Component Object Model (DCOM) [30], o Common Object Request Broker Architecture (CORBA) [21] e o Java Remote Method Invocation (JRMI) [31]. No entanto, uma limitação do conceito de RPC e das plataformas distribuídas de objectos, é a falta de suporte à replicação e a tolerância a faltas.

O modelo de negócio e as expectativas dos utilizadores da Internet tornaram a tolerância a faltas e a escalabilidade requisitos fundamentais dos sistemas computacionais. A solução mais apelativa para resolver este problema é a distribuição do serviço por vários servidores, incentivando a tolerância a faltas e a distribuição de carga. No entanto, a distribuição de um sistema aumenta a sua complexidade. Em comparação com os sistemas centralizados, os programadores necessitam de abordar um novo leque de problemas, como as falhas de componentes de rede, as falhas de servidores e ainda a sua heterogeneidade.

Uma abordagem clássica de tolerância a faltas é a replicação activa, onde todas as réplicas mantêm um estado coerente por executarem apenas operações deterministas e sempre pela mesma ordem. A replicação activa é um modelo de partilha de informação entre várias réplicas, em que todas elas têm o mesmo contexto. Este modelo pode ser revisto na máquina de estados distribuída, onde todas as réplicas convergem para o mesmo estado processando pedidos deterministas seguindo a mesma ordem.

As Plataformas de Comunicação em Grupo (GCS) [3] normalmente disponibilizam todos os mecanismos e primitivas necessários para a obtenção de uma máquina de estados distribuída, através de uma interface simples. Estas plataformas são totalmente independentes em relação a outras plataformas e cumprem todos os seus objectivos da forma mais genérica possível. Quando um programador necessita de algumas das características de uma máquina de estados distribuída para o seu sistema, pode tirar partido destas plataformas ao invés de ter que reimplementar estes conceitos. Desta forma, pode conceber o seu produto abstraindo-se da complexidade dos conceitos envolvidos e focar o que realmente importa.

1.2 Objectivos

Actualmente, existem várias DOF que implementam o conceito de RPC para o paradigma de programação orientada a objectos. Estas plataformas facilitam a partilha de informação entre processos distintos através da partilha de objectos. Assim, diferentes aplicações podem invocar métodos em simultâneo sobre a mesma instância do mesmo objecto. Contudo, estas aplicações não fornecem duas características importantes para os sistemas distribuídos: tolerância a faltas e replicação. Ou seja, se o processo que está a partilhar o objecto ficar indisponível, o objecto fica inacessível.

Todavia, existem as GCS cujo objectivo principal é a disponibilização de uma interface simples que implementa vários conceitos que podem facilitar a concretização de replicação e tolerância a faltas. Unindo estes dois modelos de plataformas, seria possível a criação de um novo género de plataformas que combina a partilha de objectos e a tolerância a faltas. De facto, a necessidade da utilização dos dois modelos em simultâneo levou ao aparecimento de diferentes plataformas com este objectivo. Existem diferentes abordagens possíveis para a concretização de uma plataforma deste género, dependendo do ponto de partida e dos objectivos. Diferentes abordagens podem permitir a solução a alguns problemas, mas inevitavelmente acabam por criar outros. Um dos problemas emergentes é a criação de uma nova interface de invocação para a utilização da nova plataforma. Este problema, pode ser denominado de problema de transparência e torna todas as implementações já realizadas sobre essa plataforma, obsoletas.

O principal objectivo deste trabalho é a realização de uma plataforma que unifique a partilha de objectos com a tolerância a faltas, mantendo a interface de uma plataforma

de partilha de objectos já existente. Desta forma, se a semântica da plataforma escolhida for totalmente respeitada, vai permitir a todas as aplicações já existentes possam usufruir da tolerância a faltas e replicação, sem qualquer alteração. Para garantir a tolerância a faltas e a replicação, o sistema deve ser constituído por vários processos. Cada processo contém múltiplas instâncias de objectos partilhados e podem ser utilizados em simultâneo por diferentes clientes. A aproximação utilizada para manter a semântica da plataforma escolhida foi a implantação de uma nova camada na pilha de comunicação, permitindo o acesso a todas as mensagens que por ela circulam. Esta nova camada respeita as interfaces já existentes, permitindo alcançar a transparência para a aplicação e a respectiva possibilidade de manipulação de mensagens. Contudo, durante a implementação deste conceito surgiram novos problemas, como por exemplo: partilhar o estado entre as várias instâncias de um mesmo objecto; garantir que os objectos partilhados são únicos; modelo para anunciar os objectos. Os objectivos secundários deste projecto passam por encontrar uma solução para estes problemas.

A plataforma resultante desta solução foi denominada de Fault-Tolerante Remote Method Invocation (FTRMI) 3. Esta plataforma utiliza o Appia [15] como GCS e utiliza o JRMI como DOF. Esta plataforma é código de camada intermédio, que acrescenta às características comuns de um DOF as propriedades que estão disponíveis num GCS. Já existem algumas plataformas que foram realizadas com o este mesmo intuito, contudo existem várias abordagens disponíveis para a solução deste problema e ainda nenhuma delas utilizou a abordagem aqui seguida. O nível de transparência obtido com esta solução permite que qualquer aplicação realizada em JRMI puro, possa usufruir da especificação da plataforma FTRMI sem qualquer alteração de código.

1.3 Contribuições

O FTRMI permite ao programador desenvolver as suas aplicações em JRMI, e partilhar todos os objectos remotos com alta disponibilidade. Estes objectos encontram-se espalhados por diferentes processos. Todos os objectos remotos que estão a ser partilhados mantêm o mesmo estado, mesmo quando são adicionados novos processos ao sistema. Todas as réplicas do sistema podem responder a pedidos vindos dos clientes, e todas elas actualizam o estado dos objectos em simultâneo, seguindo o modelo de replicação activa.

O código está disponível com uma licença aberta no site da plataforma Appia [1]. Os resultados deste trabalho foram publicados em [24, 25, 26].

[1] http://appia.di.fc.ul.pt

1.4 Estrutura do documento

Este documento está organizado da seguinte forma: O Capítulo 2 resume e compara as plataformas existentes consideradas relevantes no âmbito deste projecto e discute as suas limitações. O Capítulo 3 apresenta o desenho do FTRMI. A concretização do FTRMI e a resolução de alguns dos problemas descritos no capítulo anterior são apresentados no Cap. 4. O Capítulo 5 avalia o desempenho do FTRMI, comparando-o com uma plataforma com os mesmos objectivos.

Capítulo 2

Trabalho Relacionado

A Chamada a Procedimentos Remotos (RPC) [23] é um dos mecanismos mais utilizados de Comunicação entre Processos (IPC) [9]. O conceito de RPC surgiu com a necessidade de executar procedimentos em diferentes espaços de endereçamento, como diferentes processos. O funcionamento de um RPC é uma sequência de acontecimentos que normalmente é iniciada pelo cliente. O cliente começa por chamar um $stub$, que é uma rotina local, responsável por empacotar os parâmetros e respectiva identificação do procedimento a executar e, posteriormente desempacotar os resultados. O servidor ao receber um pedido vindo de um cliente, utiliza um $skeleton$ para escolher o procedimento a ser executado fazendo o trabalho inverso, isto é, desempacotar os parâmetros, executar o procedimento e empacotar a resposta. Quando a resposta volta ao cliente contém o resultado da execução do procedimento sem nada ter sido executado localmente, sendo necessário ao $stub$ desempacotar o resultado e retornar. Um RPC pode ser utilizado como solução em diferentes cenários como, retirar processamentos pesados do lado do cliente, invocação de serviços ou partilha de informação.

A evolução das metodologias e conceitos das linguagens de programação levou ao aparecimento da programação orientada a objectos deixando o antigo conceito de RPC obsoleto. A necessidade da continuidade de utilização do RPC levou ao aparecimento de uma nova variante deste conceito, um novo mecanismo para a obtenção do mesmo género de serviço orientado a objectos, a Invocação Remota de Métodos (RMI) [33].

2.1 Plataformas de Objectos Distribuídos

O papel das Plataformas para Objectos Distribuídos (DOF) [35] é simplificar o desenvolvimento de aplicações distribuídas que utilizem linguagens orientadas a objectos utilizando o RMI. Estas plataformas escondem, do programador, toda a complexidade necessária para lidar com a heterogeneidade dos sistemas, criando interfaces para executar invocações remotas com a mesma sintaxe de invocações locais. Contudo, não é possível ocultar totalmente a distribuição devido à existência de problemas que estão fora do con-

trolo das plataformas, por exemplo os problemas de conectividade.

As DOF, devido ao novo paradigma abordado, introduziram um novo conceito, o objecto remoto. Este novo conceito, levou à necessidade de efectuar algumas alterações ao mecanismo de RPC original. Existindo diferentes objectos remotos e múltiplas instâncias de cada um é necessário um serviço de rotulagem que diferencie as várias instâncias de uma forma unívoca. O serviço de rotulagem pode ser chamado de serviço de nomes e é responsável por indexar as diferentes instâncias de objectos. Este serviço de rotulagem, já era utilizado por algumas concretizações de RPC para indexação de dados necessários, como por exemplo, a porta utilizada. Assim sendo, um cliente começa por consultar um serviço de nomes obtendo a referência para uma dada instância de um objecto remoto, através de um nome associado ao objecto pretendido. Após a obtenção da referência, o cliente pode comunicar directamente com o detentor da instância do objecto requisitado.

2.1.1 CORBA

O Common Object Request Broker Architecture (CORBA) [21] é uma especificação que tem como objectivo definir o comportamento de um DOF separando a lógica de programação da lógica de funcionamento. O Object Request Broker (ORB) é a entidade responsável por toda a comunicação realizada entre cliente e servidor, e representa o núcleo da arquitectura CORBA. Cada objecto CORBA contém uma identificação única denominada de Interoperable Object Reference (IOR). Um IOR é utilizado pelo ORB para localizar um objecto em qualquer parte da Internet. Os IORs podem ser obtidos pelos clientes por diferentes mecanismos, por exemplo, consultando um serviço de nomes ou através de memória não volátil. É da responsabilidade do servidor criar e difundir o IOR de cada objecto que disponibiliza. A utilização de um serviço de nomes permite realizar a projecção entre, nomes utilizados na aplicação e IORs.

O CORBA separa o suporte para invocações remotas em duas componentes principais. A componente dependente da aplicação, e a componente independente da aplicação. A componente dependente da aplicação é gerada por um compilador de acordo com a interface do objecto remoto. O interface do objecto remoto deve ser previamente definida pelo programador da aplicação recorrendo à Linguagem de Definição de Interfaces (IDL). Esta componente permite a criação de $stubs$ de acordo com a interface do objecto, e ainda a criação de mecanismos necessários para a serialização de novos objectos criados no âmbito de cada projecto. Em tempo de execução, a componente dependente da aplicação utiliza um componente independente da interface do objecto e que é concretizado sob a forma de biblioteca. Este componente suporta, por exemplo, o envio de mensagens e comunicação com o ORB.

A Figura 2.1 retrata os intervenientes durante uma invocação remota no CORBA. O $stub$ do cliente e o $skeleton$ do servidor estão ambos dependentes da interface do objecto remoto. Em particular, o $stub$ apresenta uma interface para a aplicação cliente que reflecte

as características do objecto remoto. Quando é realizada uma invocação remota, o $stub$ e o $skeleton$ são responsáveis por empacotar e desempacotar todos os parâmetros de um dado método e seu retorno. O formato do empacotamento de dados está presente na especificação CORBA de modo a tornar possível a sua utilização em diferentes linguagens de programação, mantendo a interoperabilidade.

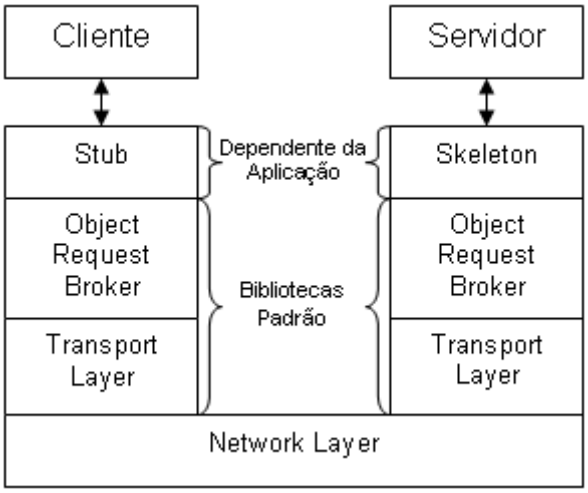

Figura 2.1: Especificação do Protocolo CORBA

2.1.2 JRMI

O Java Remote Method Invocation (JRMI) [31] é uma concretização que cumpre a especificação Distributed Java Object Model (DJOM) [29] mantendo toda a semântica proveniente do Java Object Model (JOM). O principal objectivo da plataforma JRMI é simplificar a criação de sistemas distribuídos incorporando o modelo cliente-servidor. À semelhança do CORBA, o servidor de objectos que cria os objectos é o responsável por anunciar, aos clientes, as referências que contêm a localização de cada objecto. As referências criadas pelo servidor podem ser distribuídas de várias formas, recorrendo a um servidor de nomes, ou alternativamente podem ser passadas como parâmetros de outras invocações remotas. O Registo é um serviço de nomes que se encontra num endereço e porto bem conhecido. No JRMI, e à semelhança do CORBA, o próprio servidor de Registo é um objecto remoto JRMI. A interface do servidor de Registo, interface do objecto, $stub$, e $skeleton$ são disponibilizados juntamente com a plataforma Java.

A Fig. 2.2 retrata uma aplicação distribuída utilizando JRMI. Inicialmente, o servidor de objectos cria as instâncias dos objectos e regista as referências para os objectos no servidor de nomes. O cliente obtém a referência para um objecto remoto através da consulta ao serviço de nomes, pedindo a referência associada a um dado nome. Após obtida a referência, o servidor pode invocar métodos no objecto remoto obtido.

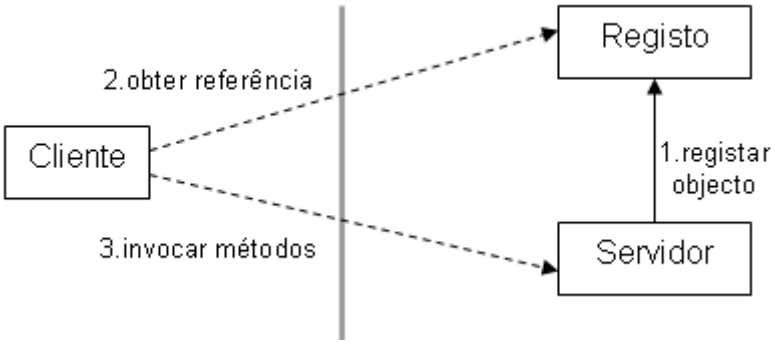

Figura 2.2: Modelo do JRMI

A arquitectura JRMI tem como benefício a adopção do JOM o que permite dissimular todos os detalhes de comunicação e indexação entre diferentes métodos e objectos de forma transparente ao programador, mantendo todo o modelo de dados descendente do Java. A criação de um serviço recorrendo à plataforma JRMI está dividida em dois passos: inicialmente a definição de uma interface Java e posteriormente a sua concretização num objecto à parte. Quando um servidor regista o objecto remoto no serviço de nomes, este está preparado para ser invocado por qualquer cliente que tenha acesso à interface criada.

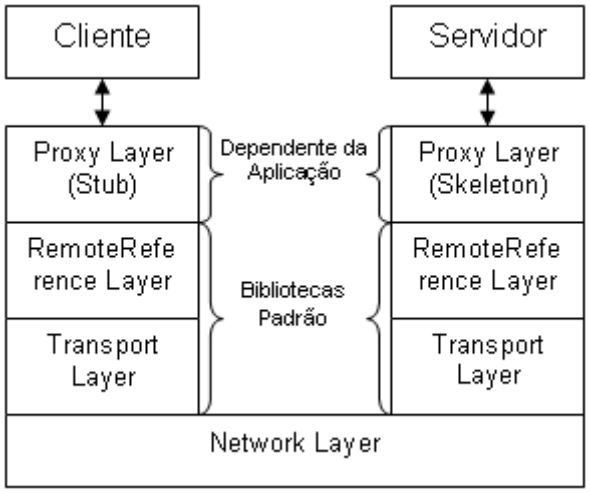

Figura 2.3: Especificação do Protocolo JRMI

A arquitectura organiza-se em três camadas retratadas na Fig. 2.3. Tal como no CORBA, os componentes $stub$ e $skeleton$ têm a responsabilidade de empacotar e desempacotar os argumentos e valores de retorno. A camada $remotereference$, não existente no CORBA, é responsável por carregar as classes em falta de objectos utilizados durante uma invocação remota, localmente se estiverem disponíveis ou remotamente. A transferência de classes pode ocorrer em simultâneo com uma invocação, isto é, se o cliente verificar que a classe de retorno de um método não está disponível, a classe pode ser enviada juntamente com a resposta. Finalmente, a camada de transporte faz a ligação entre

as máquinas virtuais Java utilizando o protocolo JRMP sobre TCP/IP.

As referências guardadas no servidor de nomes identificam uma instância do objecto remoto, associando o serviço prestado pelo objecto a um servidor em particular. Na ocorrência de uma falha do servidor, a referência torna-se inválida, mesmo que o serviço esteja disponível noutra instância do objecto. Um mecanismo não transparente para resolver este problema seria o refrescamento das referências num servidor de nomes quando fosse detectado que a referência anterior deixou de ser válida. Contudo, esta solução não é transparente para os clientes devido à necessidade de recorrer novamente ao servidor de nomes quando uma falha é detectada. No caso em que os serviços necessitem de manter o seu estado totalmente partilhado, as limitações são ainda mais nítidas uma vez que não existe suporte à replicação de estado.

2.2 Plataformas de Comunicação em Grupo

O RPC é uma boa abstracção de comunicação orientada ao modelo cliente-servidor. No entanto, há uma vasta gama de aplicações que requerem interacção directa entre grupos de processos. A comunicação em grupo tem como principal objectivo o envio de mensagens com garantia de entrega de um processo para múltiplos processos, normalmente bem conhecidos. Aplicações com requisitos de comunicação em grupo podem beneficiar deste tipo de comunicação, retirando a vantagem de algumas redes de computadores que suportam tecnologias como $broadcast$ e $multicast$ ao nível da camada de ligação de dados. Uma das dificuldades da concepção de um protocolo que utilize a camada de ligação de dados é a dificuldade da sua integração entre duas redes distintas, ou por exemplo, através da Internet. A utilização destas tecnologias através da Internet torna-se impossível, uma vez que não oferece qualquer suporte para que estas tecnologias sejam implementadas. Este problema pode ser resolvido com a criação de um algoritmo que estabeleça uma noção de grupo entre vários processos e mantenha a informação necessária para a sua comunicação, como a garantia de entrega de mensagens. Adicionalmente, um grupo deve poder variar ao longo do tempo, sendo necessário que todos os membros do grupo sejam notificados sobre as alterações de filiação. O desempenho pode ser melhorado utilizando $broadcast$ e $multicast$ quando possível.

Em cenários realistas, a ausência de limites aos atrasos de entrega das mensagens torna difícil a distinção entre um processo lento e um processo que falhou. Contudo, diversas plataformas (como por exemplo [15, 17, 32, 6]) foram desenvolvidas com o objectivo de colmatar tais problemas. Estas plataformas, denominadas de Sistemas de Comunicação em Grupo (GCS), combinam a sincronia virtual [5] com a difusão atómica. A sincronia virtual concretiza o conceito de grupo, ao qual processos se podem juntar ou sair (voluntariamente ou falhando). Cada mudança na constituição deste grupo resulta num ponto de sincronização denominado *mudança de vista*, que garante que todos os processos correc-

tos entregam as mensagem na mesma vista em que foram enviadas. A difusão atómica por sua vez, combina a garantia de entrega e a ordem total. A primeira obriga a que uma mensagem seja entregue a todos os processos correctos ou a nenhum. A segunda garante que as mensagens são entregues pela mesma ordem a todos os processos correctos, incluindo o emissor.

O objectivo principal dos GCSs é esconder dos programadores das aplicações a complexidade de concretização de algoritmos complexos de sistemas distribuídos, o que inclui as premissas acima referidas como sincronia virtual e ordem total. Exemplos de aplicações realizadas com estas plataformas são jogos distribuídos [16] e replicação de base de dados [28]. A concretização de alguns dos GCSs estende a pilha de comunicação TCP/IP criando um novo nível entre a camada de transporte e a camada da aplicação. Este conceito pode ser refinado utilizando internamente um sistema de camadas em que cada uma é responsável por concretizar um determinado serviço. O leitor interessado pode consultar [3, 34] para uma comparação interessante entre GCSs.

A máquina de estados distribuída é um conceito muito poderoso e a sua concretização não é trivial. Numa máquina de estados distribuída, um grupo de processos executa o mesmo código determinista para em conjunto com a sincronia virtual e a difusão atómica, convergirem para o mesmo estado. Contudo, as mensagens são apenas entregues a processos correctos, ou seja, se um processo falhar corre o risco de perder mensagens durante a falha e consequentemente pode não atingir o estado dos processos correctos. A sincronia virtual facilita a recuperação dos processos, uma vez que: $i)$ notifica todos os elementos correctos de alterações à sua filiação; e $ii)$ cria pontos de sincronização que podem ser usados para transferir o estado assim que um processo recupera de uma falha.

2.3 Plataformas de Replicação

Esta secção apresenta uma breve descrição de várias plataformas de replicação que foram desenvolvidas para contornar os problemas e limitações que advêm das Plataformas para Objectos Distribuídos (DOF) 2.1. As DOF apresentam vários problemas quando submetidas a cenários reais de utilização através da Internet. Os problemas fundamentais detectados foram, a transparência, escalabilidade, retro compatibilidade e replicação. A transparência é influenciada pelo grau de alterações necessário a um sistema já implementado, após algumas modificações de configuração e/ou actualização, como por exemplo mudança do protocolo de comunicação. A escalabilidade é a garantia em como um sistema distribuído pode crescer sem comprometer o desempenho. A retro compatibilidade indica-nos se uma plataforma após ser modificada e actualizada com novas versões continua compatível com versões anteriores. A replicação pode em alguns casos não ser garantida, dada a incapacidade de suporte à replicação por parte das DOFs, e consequentemente pensou aplicar-se a máquina de estados distribuída concretizada através da

utilização de um GCS. As soluções apresentadas pelas plataformas de replicação face às DOF utilizam diferentes aproximações aos diferentes problemas e limitações, e em alguns casos, surgem novos problemas. As soluções apresentadas são interessantes porque nos indicam que existem várias formas de resolver os problemas apesar dos requisitos finais serem diferentes.

2.3.1 Jgroup e ARM

O Object Group System (Jgroup) [17] é uma implementação de um GCS em Java. O Jgroup define um paradigma de programação com grupos de objectos em que cada grupo é definido por um nome. O modelo de objectos do Jgroup define as duas seguintes abstracções, o Remote Object Group (ROG) e o Replicated Remote Object (RRO), representadas na fig. 2.4. O ROG representa uma colecção de instâncias de objectos remotos que partilham a mesma interface, mas surge no cliente apenas como sendo um único. O RRO é relevante e visível apenas para o servidor e define que um objecto está a ser replicado e que pertence a um grupo.

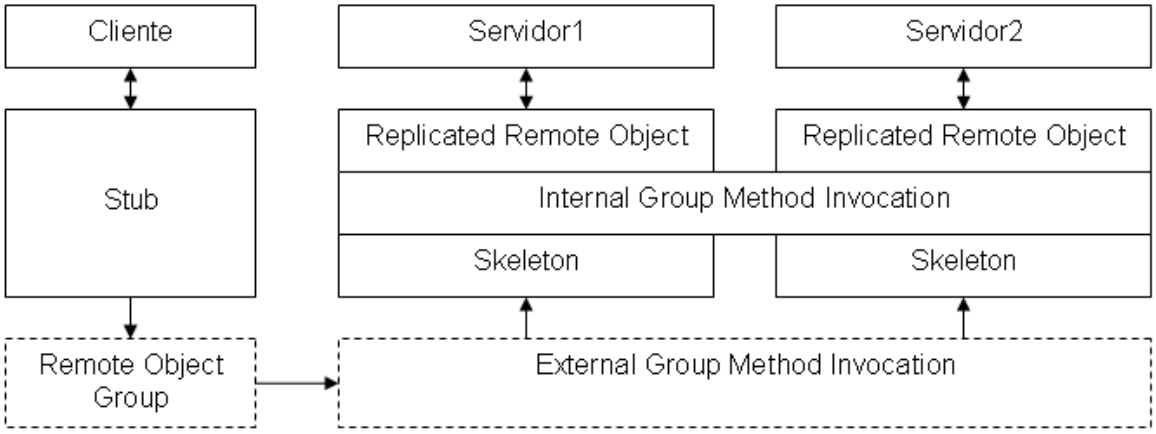

Figura 2.4: Modelo do Jgroup

O Jgroup divide-se em três serviços, Partitionable Group Membership Service (PGMS), State Merging Service (SMS) e Group Method Invocation (GMI) service. O PGMS oferece propriedades comuns dos GCS, como por exemplo, sincronia virtual. Por sua vez, o SMS fornece os mecanismos relacionados com o estado de cada réplica, como por exemplo, a sincronização do estado. O GMI é decomposto em duas componentes: Externo (EGMI) ou Interno (IGMI). O EGMI é o serviço responsável por redireccionar as invocações recebidas por cada ROG de um determinado RRO efectuando a comunicação entre o cliente e o servidor. No IGMI são utilizados mecanismos de difusão fiável para distribuir todas as invocações por todos os objectos remotos replicados. Este serviço efectua a comunicação entre os servidores. Esta arquitectura permite uma separação da comunicação entre clientes e servidores da comunicação entre servidores.

A plataforma Autonomous Replication Management (ARM) [13] é construída no topo da tecnologia fornecida pelo Jgroup e simplifica a implementação de aplicações onde são necessárias políticas de replicação específicas. Esta plataforma disponibiliza mecanismos para a distribuição de réplicas entre anfitriões e recuperação de faltas. A manutenção do número de réplicas a operar, bem como a sua disponibilidade é da responsabilidade do Replica Management (RM), um dos componentes que constitui o sistema. As réplicas são localizadas pelos clientes através da utilização de um Dependable Registry (DR). Este serviço disponibiliza um registo de nomes replicado não compatível com o Registo do JRMI original. O modelo do Jgroup combinado com a plataforma ARM disponibiliza uma plataforma de objectos distribuídos onde as invocações remotas são processadas com ordem total seguindo o modelo da máquina de estados distribuída.

Contudo, esta aplicação define novas interfaces de comunicação para facilitar a resolução de problemas como a tolerância a faltas não existindo qualquer compatibilidade com o JRMI. Nos serviços disponibilizados pelo Jgroups/ARM é necessário executar uma aplicação em todos os anfitriões do sistema. Esta aplicação é responsável por criar e manter todas as réplicas de uma forma automática.

2.3.2 FT-CORBA

O FT-CORBA [22] é uma especificação de uma arquitectura e de mecanismos de tolerância a faltas para a concretização de sistemas distribuídos de alta disponibilidade. A especificação define vários mecanismos distintos, incluindo políticas e propriedades que permitem aos programadores que utilizem plataformas compatíveis com esta especificação, a oportunidade de configuração detalhada de acordo com a sua necessidade. Ao contrário da especificação do CORBA, o FT-CORBA não define os protocolos necessários para garantir a interoperabilidade entre diferentes concretizações de um ORB. A referência para um objecto remoto FT-CORBA é realizada através de uma Object Group Reference (IOGR). Esta referência é criada e mantida desde a criação de um grupo até à sua destruição e a composição deste grupo pode mudar durante o seu tempo de vida. Um grupo é criado e mantido inteiramente pela infra-estrutura do FT-CORBA.

A arquitectura do FT-CORBA é composta por três organismos: entidade de redundância, detecção de faltas e recuperação de faltas. A entidade de redundância é responsável por todos os detalhes relativos à gestão e manutenção de um grupo. Um grupo é constituído por várias réplicas que partilham o mesmo objecto replicado. Um objecto replicado pelo FT-CORBA é constituído por diferentes instâncias de objectos CORBA que partilham a mesma interface. Assim sendo, cada objecto CORBA tem um IOR e cada objecto FT-CORBA tem um IOGR que representa um conjunto de IORs correspondentes ao grupo de réplicas que mantêm um objecto. Para além da gestão interna de manutenção de grupos, a criação de novos IOGR também é da competência da entidade de redundância. A entidade de recuperação de faltas faz o histórico de operações em cada objecto permi-

tindo a sua recuperação para um estado especifico pretendido. Esta entidade também está preparada para uma troca de estado completa, e portanto deve saber serializar um objecto.

A entidade que detecta as faltas deve estar presente em todas as réplicas que constituem o sistema e fornece a todos os seus componentes informação de monitorização sobre as diferentes réplicas. A entidade responsável por detectar as faltas está dividida em duas componentes, o detector de faltas e o notificador de faltas. Estas duas componentes estão interligados e mantêm um estado em comum de todo o sistema permitindo a qualquer componente deste subscrever eventos de notificação de acordo com a sua funcionalidade. No detector de faltas podem ser adicionados novos componentes a serem monitorizados, e no caso do notificador de faltas, podem ser subscritos os eventos correspondentes à monitorização. Para detecção de faltas o detector de faltas segue o modelo $pull$, ou seja, verifica periodicamente se um componente falhou através da ausência de resposta.

O FT-CORBA permite definir três tipos distintos de objectos replicados: sem estado, passivos e activos. Nos objectos sem estado é assumido que não é necessário manter o contexto entre duas invocações, e por consequência não é necessário a troca de estado entre réplicas. Os objectos passivos assumem que existe uma instância do objecto pertencente ao grupo que é responsável por processar todos os pedidos criando pontos de restauro no histórico de invocações que podem ser exportadas e invocadas nas restantes réplicas. Os objectos activos assumem que é necessário manter o contexto entre duas invocações e por consequente a invocação de operações deve ocorrer pela mesma ordem em todas as instâncias de um objecto de um dado grupo.

Contudo, esta especificação requer modificações de raiz aos ORBs existentes e a eventuais aplicações que utilizam a especificação do CORBA, não existindo qualquer retro compatibilidade.

2.3.3 Interoperable Replication Logic

O Interoperable Replication Logic (IRL) [12] é uma implementação de um tipo específico de lógica de replicação que satisfaz propriedades como replicação activa e tolerância a faltas de uma forma não intrusiva para o CORBA. A lógica de replicação onde são redefinidas interfaces foi centralizada num único componente, contrariamente a outras implementações. No IRL, o cliente interage com vários servidores distintos, e cada servidor contém objectos independentes. Consequentemente, cada servidor disponibiliza a mesma interface para cada objecto, e o IRL tira partido desta interface para disponibilizar um serviço diferente do serviço original. O posicionamento desta aproximação numa arquitectura CORBA é exactamente entre o $stub$ ou o $skeleton$ e o ORB 2.1. O IRL é composto por três componentes principais: IRL Core, SmartProxy e ServerRunTimeSupport. O IRL Core é composto por vários objectos CORBA comuns colocados sobre um ORB genérico. Estes objectos contêm grupos de IORs em que cada grupo corresponde a um único objecto e suas réplicas, mantendo a separação da lógica de replicação dos ob-

jectos da aplicação. O próprio IRL Core necessita de ser replicado utilizando replicação passiva para que ele próprio não seja um ponto único de falha. O SmartProxy é o componente acima do ORB do cliente que se responsabiliza por esconder a replicação activa, redireccionando todos os pedidos para a réplica primária do IRL Core, que posteriormente é responsável por ordenar todos os pedidos com ordem total. O ServerRunTimeSupport disponibiliza vários objectos CORBA que são necessários à execução da aplicação. Estes objectos permitem a manutenção do sistema de detecção de faltas e dos mecanismos de recuperação para ocultar os detalhes da replicação activa. O IRL é compatível com qualquer implementação genérica da especificação do CORBA. Contudo, a centralização do IRL Core na réplica primária pode apresentar problemas de desempenho.

2.3.4 Jini

O Jini [1, 18] tem como objectivo desenhar especificações para uma vasta gama de serviços que incorporam novos paradigmas, mantendo a segurança e simplificando a configuração para a composição de aplicações flexíveis e distribuídas. A meta final é composta pela definição de várias interfaces com diferentes objectivos e uma possível implementação para todos os serviços elaborados. Melhorar ou corrigir alguns problemas detectados no JRMI, como por exemplo a invocação remota segura que surge com mais verificações no cliente. O suporte para $proxies$ de comunicação flexíveis e ainda de diferentes fornecedores de transporte por cada objecto, unificando vários paradigmas de comunicação numa simples interface para facilitar o trabalho do programador.

A arquitectura Jini divide-se em três categorias: infra-estrutura, serviços e modelo de programação. A infra-estrutura é o conjunto de componentes que permite construir um sistema Jini, enquanto os serviços são as entidades responsáveis por cada um dos serviços a disponibilizar. O modelo de programação é o conjunto de interfaces que define um serviço possibilitando ao programador construir novos serviços ou diferentes implementações para os existentes. Todas as implementações de serviços que o Jini oferece por omissão respeitam o mesmo modelo de programação. A infra-estrutura debruça-se sobre segurança, limitações ao nível da invocação e políticas de segurança dinâmicas. Os diferentes modelos de comunicação disponibilizados são: Lookup Service (reggie), Transaction Manager Service (mahalo), Lease Renewal Service (norm), Event Mailbox Service (mercury), Lookup Discovery Service (fiddler); e JavaSpaces Service (outrigger) [1].

2.3.5 Transparent Consistent Replication of Java RMI Objects

O Aroma [20] é uma plataforma de código intermédio transparente para uma aplicação JRMI que fornece tolerância a faltas. O Aroma escuta toda a camada de transporte da

[1] http://river.apache.org/user-guide-basic-river-services.html

pilha TCP/IP, ou seja, todos os dados enviados pelo cliente e pelo servidor são interceptados. As mensagens que são relevantes são difundidas pelas respectivas réplicas utilizando o Totem [19], um GCS, para obtenção de ordem total. A ordem total é relevante para obtenção de uma ordem de processamento de pedidos perante todas as réplicas utilizando a máquina de estados distribuída.

O sistema Aroma é composto por quatro componentes: Interceptor, Parser, Message Handler e Multiplexer. O Interceptor encarrega-se de escutar toda a camada de transporte na pilha TCP/IP incluindo as operações: $bind$, $listen$, $read$, $write$ e tráfego. O Parser em conjunto com o Interceptor identifica apenas as mensagens capturadas que são específicas do JRMI filtrando-as de entre todas as outras. O Message Handler em conjunto com o Parser processa apenas mensagens JRMI e finalmente prepara-as para introduzir no GCS através do Multiplexer. O Message Handler é de todas as camadas a mais complexa. Esta camada mantém um projecção entre réplicas e seus respectivos grupos, difunde os pedidos vindos do JRMI para todas as outras réplicas e é responsável por manter os objectos remotos coerentes. A coerência pode ser assegurada de duas formas, activamente e passivamente. Na replicação activa, a ordem de processamento dos pedidos em todas as réplicas é a mesma. Na replicação passiva, quando uma única réplica processa todos os pedidos e posteriormente envia uma listagem dos pedidos processados, com a ordem respectiva, para as outras réplicas. Cada mensagem enviada é etiquetada univocamente pelo Message Handler para prevenir o processamento de mensagens replicadas. O servidor de nomes necessário para a projecção de nomes em objectos remotos pode ser um ponto de falha. No entanto, para o Aroma é apenas mais um objecto remoto portanto a sua replicação é trivial. É imposto um modelo de tarefa única para apenas ser processado um pedido de cada vez eliminando assim problemas de coerência resultantes do escalonamento das tarefas em diferentes réplicas. Adicionalmente, o Aroma não suporta invocações que envolvam várias tarefas. O sistema é transparente para uma aplicação JRMI sendo apenas incómodo e ineficiente o facto de existir uma aplicação que lê todo o tráfego de saída e entrada incluindo no servidor e no cliente.

Contudo, o maior problema criado com esta aproximação é a reconciliação de estados após a presença de uma nova replica sem estado. Uma possível aproximação para a resolução deste problema seria a criação de um histórico de pedidos executados em todos os objectos disponibilizados, que permitisse a um novo objecto atingir o estado das restantes réplicas.

2.3.6 Efficient Replicated Method Invocation in Java

O objectivo do Efficient Replicated Method Invocation in Java [10] é construir uma aplicação eficiente e semelhante ao JRMI de forma a permitir replicação de objectos respeitando as premissas da máquina de estados distribuída. Nesta solução foi utilizado um compilador alterado que pré processa o código em Java. As duas interfaces disponibiliza-

das Root e Node têm um papel semelhante à interface Remote fornecida pelo JRMI. Em tempo de compilação são gerados $wrappers$ para os objectos remotos e todos os métodos são ingenuamente analisados. A análise realizada permite rotular os métodos como de escrita ou só de leitura olhando apenas para atribuições directas. Nas situações em que não é trivial determinar que um método não é só de leitura a plataforma analisa-o em tempo de execução. Nestes casos, todos os métodos são inicialmente executados como só de leitura e caso seja executada alguma operação de escrita é lançada uma excepção. A excepção é colocada no código durante o pré processamento em tempo de compilação. Esta excepção aborta a operação em causa, e volta a executá-la com permissões de escrita. Uma execução de um método só de leitura é bastante diferente de uma execução de escrita. A coerência do objecto não é afectada quando a execução é abortada porque apenas são abortadas execuções só de leitura. Esta grande distinção entre métodos só de leitura e de escrita é importante para melhorar o desempenho. Quando um método só de leitura é executado o servidor em causa pode retornar o resultado tendo em conta o objecto que possui localmente sem contactar outras réplicas. No caso de um método de escrita a operação é enviada para todas as réplicas utilizando ordem total e é executada depois aquando da recepção. As réplicas apenas possuem uma tarefa de execução de operações recebidas por ordem total, de forma a garantir uma ordem e todas as réplicas atingirem o mesmo estado. A ferramenta utilizada para difusão de mensagens em ordem total é o Panda [2] e a serialização de objectos é realizada por um mecanismo eficiente disponibilizado pelo compilador utilizado.

Esta solução é eficiente e mantém o desenho do JRMI de uma forma replicada. É necessário definir os objectos de acordo com as interfaces disponibilizadas mas em contrapartida existe distinção entre operações só de leitura e escrita. Contudo, toda a estrutura do JRMI foi ignorada, não existindo qualquer possibilidade de compatibilidade entre ambas.

2.3.7 Filterfresh: Hot Replication of Java RMI Server Objects

O Filterfresh [4] é uma implementação que partilha as interfaces que constituem o JRMI. O JRMI original é constituído por: cliente, servidor, e serviço de nomes. O Filterfresh disponibiliza uma implementação diferente com suporte a tolerância a faltas para cada uma das três componentes. No cliente, o componente alterado foi o $stub$, foi preparado para comunicar com servidores distintos, por forma a garantir a entrega de um pedido. No servidor foi implementada uma nova classe que permite interceptar todos os pedidos vindo de um cliente e desta forma ordená-los antes do seu processamento. A aproximação utilizada para disponibilizar um serviço tolerante a faltas, foi a simulação de uma máquina de estados distribuída utilizando ordem total. No Filterfresh foi criada uma nova identidade, denominada GroupManager que gere os detalhes de comunicação e ordenação de pedidos de acordo com o GCS utilizado. O servidor de registo foi implementado de raiz

para garantir que também faz parte do mesmo grupo de comunicação disponibilizado no GroupManager. O mecanismo de referências que o JRMI utiliza para localizar objectos também foi ligeiramente alterado, de forma a ser possível alterar o servidor para o qual o Filterfresh está conectado para um outro distinto. Esta solução requer a adição de classes no lado do cliente e no lado do servidor e uma nova implementação do registo. Contudo, esta solução respeita toda a interface do JRMI porque apenas são estendidas interfaces já existentes na sua especificação.

2.4 Comparação de Plataformas

A tabela 2.1 compara diferentes aproximações para a resolução dos problemas de replicação e de tolerância a faltas para as DOF. Na coluna à esquerda encontram-se as plataformas presentes na comparação. Na primeira linha estão as propriedades presentes na comparação. A plataforma estendida indica qual é a DOF em que a plataforma que está a ser comparada se baseou para a sua implementação. A transparência é a propriedade que qualifica a que nível foram feitas as alterações. Se a plataforma se baseia em uma DOF e as alterações efectuadas para atingir o objectivo são alterações de arquitectura faz com que todas as implementações já existentes relativas à determinada DOF deixam de ser compatíveis. Por outro lado, se a plataforma apenas contribuir para alcançar os objectivos sem alterações à arquitectura da DOF, consegue alguma retro compatibilidade para implementações anteriores, mantendo a total transparência à sua execução. A replicação activa e passiva são duas propriedades mutuamente exclusivas em tempo de execução, ou seja, a plataforma só pode estar configurada para utilizar uma delas de cada vez. Contudo, algumas implementações suportam ambas. O serviço de registo replicado indica-nos se a plataforma fornece algum mecanismo novo para a DOF a ser estendida, de forma a manter o serviço de registo tolerante a faltas. A transferência de estado é relevante para que todas as réplicas da plataforma consigam convergir para um único estado. Todavia, dependendo da forma como as plataformas atacaram cada solução para resolver os problemas, em algumas soluções esta propriedade não está disponível devido à dificuldade da sua obtenção.

As duas propriedades mais difíceis de garantir em simultâneo são a transparência e a transferência de estado. No caso da transparência podem ser utilizadas técnicas de baixo nível para facilitar sua obtenção. Por exemplo, a abordagem praticada pelo Aroma utiliza uma forma de escutar todo o tráfego que circula em toda a pilha TCP/IP, filtrando apenas o que lhe interessa. Contudo, esta abordagem dificulta a implementação da transferência de estado de uma forma transparente à DOF, neste caso o JRMI. O IRL por sua vez, consegue obter a transparência e a transferência de estado em simultâneo devido à liberdade oferecida pelo CORBA. Esta plataforma analisou na pilha de comunicação do CORBA qual seria a melhor camada para ser alterada, de forma a conseguir a transparência para o

Tabela 2.1: Comparação entre plataformas

	Plataforma Estendida	*Transparência*	*Replicação Activa*	*Replicação Passiva*	*Registo Replicado*	*Transferência de Estado*
Jgroup/ARM			•		•	•
FT-CORBA	CORBA		•	•	•	•
Interoperable Replication Logic	CORBA	•	•		•	•
Jini	JRMI		•	•	•	•
Transparent Consist Replication of Java RMI Objects	JRMI	•	•	•	•	
Efficient Replicated Method Invocation in Java			•		•	•
Filterfresh	JRMI		•		•	•

cliente e para o servidor. A camada escolhida foi o ORB, que mistura a lógica de objectos com a camada de rede. O CORBA permite configurar qual o ORB que vai ser utilizado e desta forma o IRL consegue alcançar o seu objectivo. Contudo, esta abordagem fica limitada ao facto de ser necessário a existência de uma réplica primária para a gestão de comunicação entre todos os ORBs de todas as réplicas. O JRMI ainda não dispõe de nenhuma plataforma que satisfaça estas duas propriedades em simultâneo, ou seja, quando utilizamos uma das plataformas disponíveis é necessário optar pelo transparência ou pela transferência de estado.

Capítulo 3

Fault-Tolerante Remote Method Invocation

O objectivo principal da plataforma Fault-Tolerante Remote Method Invocation (FTRMI) é fornecer um serviço de replicação activa de objectos. O FTRMI disponibiliza este serviço de uma forma totalmente transparente para uma aplicação Java Remote Method Invocation (JRMI) [31], podendo ser utilizada sem qualquer recompilação de código. A plataforma beneficia da ubiquidade do JRMI, estendendo-o com novas bibliotecas que aumentam as suas funcionalidades, nomeadamente a tolerância a faltas e a recuperação após uma falha. Um aspecto interessante do FTRMI é a sua transparência total para os clientes JRMI, uma vez que as novas bibliotecas apenas são necessárias no servidor.

A arquitectura de comunicação do FTRMI utiliza exclusivamente o protocolo JRMI entre o cliente e o servidor. Por outro lado, nos servidores foi inserida uma nova camada de comunicação que permite a troca de mensagens entre eles, de forma a concretizar uma máquina de estados distribuída. Quando um pedido chega a uma das réplicas, é reencaminhado para a nova camada de comunicação e difundido com ordem total por todas as réplicas disponíveis, para garantir a coerência. Os pedidos, já ordenados, são executados e a resposta é enviada de volta ao cliente por JRMI. A nova camada de comunicação utiliza o Appia [15] como plataforma de suporte à comunicação.

3.1 Descrição da Arquitectura

O estudo de plataformas revelou os pontos fracos e os pontos fortes de algumas plataformas realizadas com o mesmo objectivo do FTRMI. Desta forma, a plataforma FTRMI pretende tirar partido de todos os pontos fortes como, transparência, replicação e sincronização de estado, de forma a satisfazer o maior número de requisitos eliminando os pontos fracos.

Um sistema tolerante a faltas deve cumprir várias propriedades como inexistência de um ponto único de falha. A falta deve ser localizada para uma possível substituição do componente que falhou e a falta não se deve alastrar a outros componentes do sistema.

Deste modo, uma das formas de implementar a tolerância a faltas num sistema distribuído, como o JRMI, é utilizando a replicação. A aproximação por replicação permite que em caso de uma falha exista outro componente que saiba reagir e responder correctamente a um determinado serviço. A replicação pode ser categorizada em três tipos, activa (todas as réplicas processam pedidos e todas mantêm o mesmo estado), passiva (uma das réplicas processa pedidos e envia a ordem de processamento para as restantes réplicas utilizando pontos de sincronização) e semi-activa (uma das réplicas define a ordem de processamento e informa as restantes para processarem o pedido). No caso do FTRMI, optou-se pela replicação activa.

A arquitectura do JRMI é composta por três componentes principais: o cliente, o servidor e o registo. O cliente localiza um servidor para um determinado serviço através de um registo, e posteriormente executa o serviço no dado servidor. Assim sendo, para tornar esta aplicação tolerante a faltas é necessário que tanto o servidor como o registo sejam replicados para que nenhum deles constituíam um ponto único de falha.

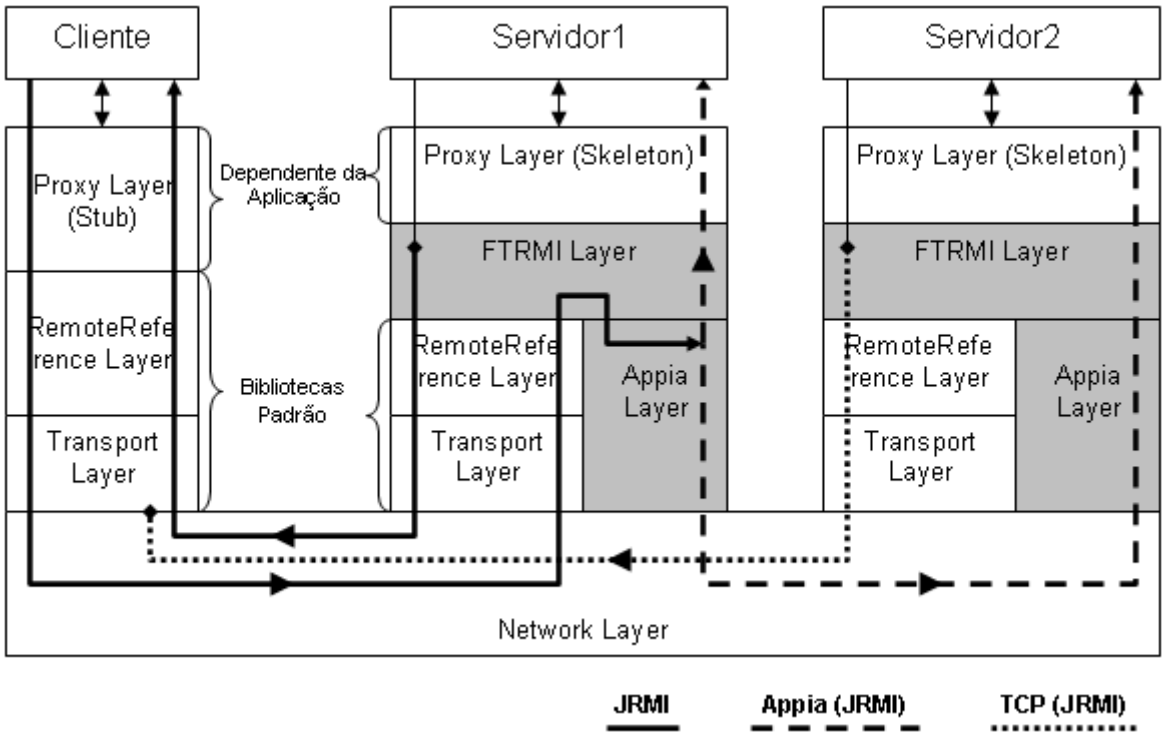

Figura 3.1: Arquitectura do FTRMI

A Fig. 3.1 retrata a arquitectura de uma aplicação distribuída utilizando o FTRMI. A chave para adquirir a transparência do FTRMI em relação ao JRMI é a reutilização de componentes. Isto é, o FTRMI não altera a camada directamente dependente da aplicação, nem as bibliotecas padrão. Como resultado, qualquer aplicação programada com a tecnologia JRMI é totalmente convertível em FTRMI sem qualquer alteração no código fonte. O FTRMI é implementado numa nova camada de comunicação no lado do servidor, co-

locada entre as bibliotecas padrão e as camadas dependentes da aplicação. Isto permite que durante a execução de uma aplicação que utilize o FTRMI, o cliente utilize as bibliotecas JRMI fornecidas com as distribuições comuns da plataforma Java. No lado do servidor, algumas bibliotecas padrão do JRMI são substituídas por bibliotecas desenvolvidas no âmbito deste projecto. A substituição é exclusivamente realizada por mudanças dos parâmetros do carregador de classes.

As setas da Fig. 3.1 representam a execução de uma invocação remota utilizando o FTRMI. No lado do cliente, uma invocação regular JRMI é preparada e enviada para a rede. Quando este pedido é entregue à camada do FTRMI no lado do servidor, o pedido é difundido para todos os servidores utilizando um canal Appia. O canal Appia utilizado pelo FTRMI está configurado para disponibilizar sincronia virtual e difusão atómica permitindo desta forma a concretização de uma máquina de estados distribuída. À medida que cada servidor recebe o pedido (incluindo o servidor que enviou o pedido), a camada do FTRMI reencaminha-o directamente para o $skeleton$. Quando o FTRMI reencaminhou o pedido para o canal de comunicação, guardou a informação necessária relativamente ao pedido, para poder interceptar o pedido depois de este ter sido processado. Na perspectiva do $skeleton$ e da aplicação, a invocação recebida é semelhante a qualquer invocação remota realizada pelo JRMI.

Durante o processamento do pedido, todas as réplicas invocam o método correspondente no objecto local e obtêm o resultado. Este resultado fica directamente na posse do FTRMI. O FTRMI intercepta todos os pedidos processados que partiram do próprio servidor, e desta forma, o servidor que originalmente processou o pedido responde de volta ao cliente com o protocolo JRMI. Os restantes servidores que também processaram o pedido mas não têm ligação directa ao cliente, preparam uma resposta ao nível do Transmission Control Protocol (TCP) idêntica à resposta que sai do servidor que processou o pedido. A informação necessária para preparar a resposta ao nível do TCP é difundida juntamente com o pedido inicial em ordem total, ou seja, o servidor quando atende um pedido e o reencaminha para o canal de comunicação, adiciona-lhe toda a informação necessária.

No lado do cliente, são recebidas N respostas consoante o número de servidores existentes, no caso em que nenhuma é perdida. Todas as respostas são idênticas, apesar de terem sido enviadas por servidores diferentes a resposta enviada ao nível do TCP permite alterar os dados de origem da resposta. Devido à semelhança das respostas, a camada de comunicação de transporte que neste caso é o TCP, encarrega-se de ignorar de uma forma transparente todas as respostas idênticas, assumindo que o mesmo servidor enviou N vezes a mesma resposta.

3.1.1 Replicação do Registo

O FTRMI promove a disponibilidade dos serviços criando múltiplas réplicas dos objectos. Contudo, assumindo que todas as réplicas têm a mesma probabilidade de falhar, os ganhos

de disponibilidade são nulos se um cliente apenas puder contactar uma das réplicas. A solução que satisfaz estes requisitos é a utilização de múltiplas referências que apontam para vários objectos remotos, e em conjunto utilizar um serviço tolerante a faltas que as disponibilize. A criação de um registo distribuído tolerante a faltas não é possível sem alterar o modelo utilizado pelo JRMI em que o registo guarda uma referência exacta para a localização de um único objecto. Infelizmente, a alteração do modelo de referências do JRMI obrigaria à mudança da interface de comunicação entre o cliente e o registo e anularia os requisitos de transparência da plataforma.

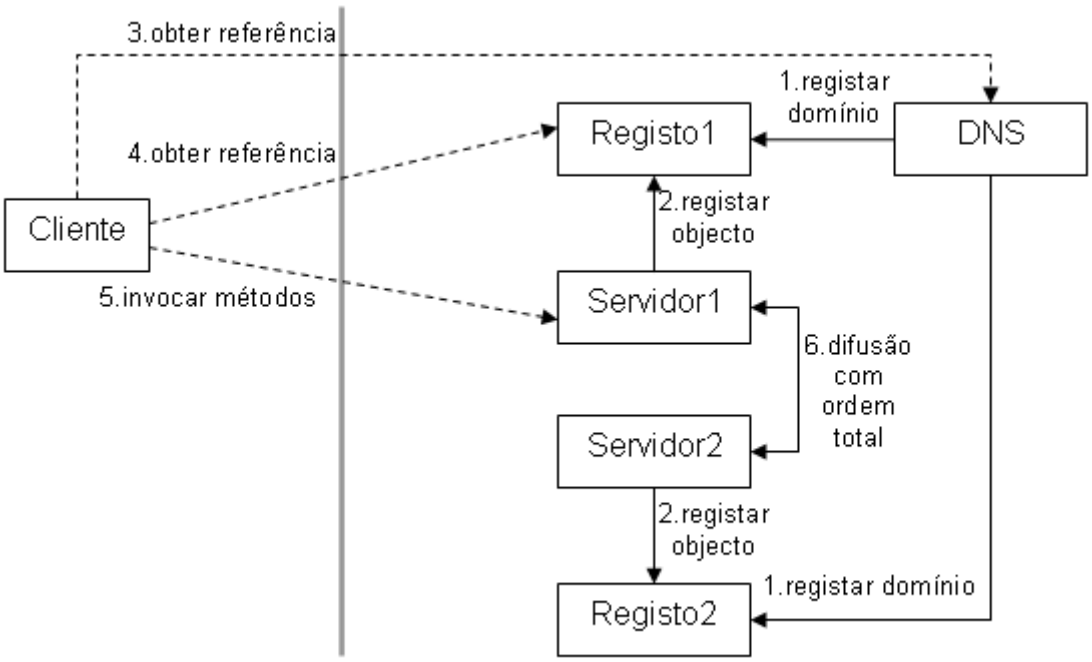

Figura 3.2: Modelo do FTRMI

A solução de compromisso adoptada pelo FTRMI está retratada na Fig. 3.2. Assumindo que cada servidor de objectos cria o seu serviço de registo JRMI, no qual regista todos os seus objectos remotos, obtemos N servidores de objectos para N servidores de registo. Esta solução é realizada recorrendo apenas a bibliotecas originais do JRMI mantendo o tipo de referências utilizado. O acesso aos diversos registos é assegurado pelo algoritmo de distribuição de carga do Domain Name Service (DNS). Este algoritmo é denominado de $roundrobin$ e tendo em conta uma lista de IPs associados a um nome, para cada pedido devolve um IP distinto atribuído sequencialmente, ao chegar ao fim da lista volta para o inicio.

A utilização do serviço de DNS pode ser vantajosa e ao mesmo tempo desvantajosa. A sua utilização permite fornecer um mecanismo de balanceamento de carga que é disponibilizado pelo sistema rotativo de endereços para um dado nome. Contudo, quando um cliente perde a referência para um objecto devido a uma falha de um servidor é necessário

voltar a consultar um novo registo. A resposta obtida do serviço de DNS para o mesmo domínio pode ser a mesma, devido ao sistema de $cache$, ou pode ser uma referência que já não é válida, devido a falhas em outros servidores.

Como trabalho futuro, foi planeado aplicar as capacidades de tolerância a faltas do FTRMI ao serviço de registo. A concretização totalmente transparente para o cliente não é trivial devido à orientação cliente-servidor do JRMI.

Uma nova implementação de um serviço de registo seria necessário para manter a transparência e fornecer um registo tolerante a faltas. Assumindo que todos os servidores que fornecem os objectos remotos e replicados se encontram no mesmo grupo virtual criado pelo Sistemas de Comunicação em Grupo (GCS) 2.2. Podemos também assumir que todos os servidores trocariam as referências correspondentes a todos os objectos remotos partilhados, ou seja, existiriam N referências por objecto remoto em que N representa o número de servidores e o número de referências distinto para cada objecto. Toda esta informação poderia ser armazenada e actualizada em tempo de execução.

Com esta informação armazenada e disponível entre processos de um grupo virtual, seria possível criar um serviço de registo tolerante a faltas disponível através de duas interfaces diferentes. A primeira interface seria reservada aos servidores e permitia o registo de diferentes referências para o mesmo nome, bem como a sua remoção. A segunda interface seria exactamente igual à disponibilizada pelo JRMI e retornava apenas uma referência por cada nome. Este serviço de nomes, para garantir o balanceamento de carga poderia utilizar o modelo $roundrobin$ entre as referências disponíveis para cada nome quando responde aos clientes.

As diferenças entre esta solução, e a solução já oferecida pelo FTRMI, seria o facto de nesta solução existir um estado global de todas as referências actuais partilhado por todas as réplicas, ao contrário da implementação actual em que cada servidor possuiu um servidor de registo com as suas referências. Com a nova solução, o problema de devolver referências inválidas seria eliminado. Note-se que quando um servidor abandona o grupo virtual as referências possuídas por este, seriam removidas do servidor de registo. A localização das réplicas de servidores de registo poderiam da mesma forma estarem associadas a um domínio DNS. Para este serviço funcionar correctamente, é necessário assumir que um cliente consulta o último servidor de registo quando perde o contacto com a réplica onde estava a executar operações. No caso de o último servidor de registo também falhar, seria necessário consultar novamente o serviço de DNS até encontrar um servidor de registo válido.

3.2 Tolerância a Faltas

A plataforma FTRMI tem como principal objectivo fornecer um serviço tolerante a faltas, ou seja, em que as falhas de um componente são atenuadas pelos restantes. Idealmente,

espera-se que todas as falhas ocorridas no serviço sejam totalmente escondidas do cliente. O FTRMI é composto por diferentes servidores que mantêm as instâncias de todos os objectos remotos no mesmo estado. Desta forma, quando um dos servidores falha, um cliente pode continuar a utilizar o serviço ligando-se a outro servidor.

Para fornecer um serviço com estas características surgem vários problemas. Por um lado, os clientes JRMI não estão preparados para em caso de falha se ligarem automaticamente a um novo servidor. Por outro, Um servidor JRMI não está preparado para utilizar técnicas de sincronização com outros servidores de forma a sincronizarem os seus estados nem existem pontos de sincronização possíveis definidos pela plataforma.

3.2.1 Ordenação de Pedidos

Os vários servidores que constituem a aplicação FTRMI devem partilhar os mesmos objectos remotos em tempo-real. Uma das melhores aproximações para fornecer um serviço tolerante a faltas é a utilização de uma máquina de estados distribuída. O modelo de replicação que melhor representa uma máquina de estados distribuída é a replicação activa. Uma máquina de estados distribuída pode ser concretizada utilizando as funcionalidades fornecidas por um GCS de forma a abstrair os detalhes da sua implementação e dirigir os esforços a resolver os problemas de adaptação da nossa plataforma. A plataforma GCS utilizada no âmbito deste projecto foi o Appia, que suporta múltiplos algoritmos de ordem total e é implementado em Java.

Uma das formas de conseguir implementar uma máquina de estados distribuída é a utilização de algoritmos de ordem total. Para tal, existem vários tipos de algoritmos de ordem total, mas para esta plataforma é importante focar os seguintes. O algoritmo de ordem total regular diz-nos que se uma mensagem é entregue por um processo correcto então essa mensagem é entregue por todos os processos correctos restantes. O algoritmo de ordem total uniforme diz-nos que se uma mensagem é entregue por um processo correcto ou incorrecto então essa mensagem é entregue por todos os processos correctos restantes. Em ambos os algoritmos está presente que todas as mensagens são entregues pela mesma ordem a todos os processos.

No âmbito do FTRMI é necessário utilizar o algoritmo de ordem total uniforme que garante melhor qualidade de serviço, podendo também existir uma quebra no desempenho. Todos os servidores FTRMI preparam uma resposta para enviar ao cliente, e desta forma, se fosse utilizado o algoritmo de ordem total regular poderiam existir respostas diferentes consoante algum dos servidores falhasse.

3.2.2 Sincronização de Estado

O FTRMI assume um modelo de faltas por paragem [8] e portanto quando um servidor recupera, os objectos regressam ao estado com que foram iniciados da primeira vez. Con-

tudo, os detectores de faltas utilizados pelos GCSs podem facilmente confundir faltas com problemas de rede. No último caso, um servidor pode regressar ao grupo apenas com um estado desactualizado.

A opção de concretização do FTRMI evita este problema tratando ambos os casos da mesma forma. A sincronia virtual desempenha um papel fundamental no processo de escolha do estado, permitindo a criação de pontos de sincronização a cada alteração da filiação do grupo. O algoritmo de escolha do estado mais actualizado é executado em duas fases para evitar que todos os servidores enviem todas as instâncias de todos os objectos que possuem para todos eles.

Quando existe uma mudança de vista, todos os servidores enviam com ordem total, a vista, a versão, e o $hashcode$ do estado actual de cada objecto. Cada servidor fica em espera até coleccionar toda a informação de todos os servidores e inclusive dele próprio. Após todos os servidores terem a informação de todos os restantes, é possível decidir deterministicamente quais deles possuem os estados mais actualizados de cada objecto. Os estados dos objectos em que algum servidor não se encontra na última versão, são difundidos apenas pelo servidor que contém o estado mais recente do respectivo objecto evitando a duplicação de informação. Todos os servidores esperam por receber todos os estados em falta, e só depois prosseguem para as execuções normais. O tempo de espera evita que sejam processados pedidos vindos dos clientes antes de existir localmente a última versão dos objectos e ainda salvaguarda o algoritmo de troca de estado, que pode ser reinicializado com uma nova mudança de vista. Os pedidos recebidos dos clientes concorrentemente com sincronização, são guardados numa lista, e após o estado de todos os objectos ser actualizado são difundidos por ordem de chegada, como se tratasse de um pedido normal. A optimização do algoritmo permite que, se existirem três objectos, dois podem ser enviados por um servidor, e o restante por um servidor distinto dependendo de quem possui a última versão. Além disso, quando um dos servidores recebe o estado de um objecto, substitui o antigo estado pelo novo, mesmo que contenha a ultima versão para desta forma garantir a coerência total. O estado é difundido pelo processo eleito utilizando as técnicas padrão de serialização de objectos disponibilizadas pelo Java, o que assegura a transparência do sistema.

3.2.3 Transparência de Faltas

A Fig. 3.3 apresenta o diagrama de mensagens de uma invocação remota. Na primeira etapa, o cliente envia o pedido para uma réplica utilizando JRMI e fica em espera de uma resposta. Quando o servidor recebe o pedido, após a primeira etapa, reencaminha o pedido para a plataforma Appia. Todos os pedidos colocados no Appia levam consigo informação acerca da conexão TCP entre o cliente e o servidor JRMI que inicialmente atendeu o pedido, nomeadamente, o IP origem, o IP destino, a porta origem, a porta destino, e os números de sequência (ACK e SEQ). A informação adicional colocada na

plataforma Appia juntamente com o pedido é relevante para a geração da resposta. Se esta réplica falhar antes de difundir o pedido pelo Appia, o cliente obtém uma excepção e deve prosseguir com a tentativa de localizar uma nova réplica utilizando o registo.

Na terceira etapa, se a réplica que está directamente conectada ao cliente falhar podem ocorrer várias situações dependendo do algoritmo de difusão total utilizado. Se o algoritmo utilizado for de ordem total regular, uma das réplicas pode decidir a ordem de entrega de uma mensagem e inevitavelmente pode processá-la. No caso, de esta réplica ser a que está a comunicar directamente com um cliente, pode até gerar uma resposta para o cliente, e só posteriormente tentar comunicar aos restantes constituintes do grupo virtual a ordem de processamento que definiu para as mensagens. Desta forma, utilizando este algoritmo pode fazer com que o cliente fique consciente em como um pedido foi executado correctamente, mas na realidade foi apenas executado no servidor que lhe forneceu a resposta que posteriormente pode falhar sem comunicar aos restantes servidores que a mensagem foi entregue. Nesta situação, a plataforma FTRMI não cumpre os requisitos referidos em 3.2. Contudo, o algoritmo de ordem total uniforme garante-nos que a mensagem é entregue a todos os servidores se foi processada por algum deles. A utilização do algoritmo de ordem total uniforme é fundamental para garantir que os requisitos são cumpridos, apesar deste algoritmo apresentar um acréscimo em número de mensagens e respectivas latências que podem prejudicar o desempenho global da plataforma.

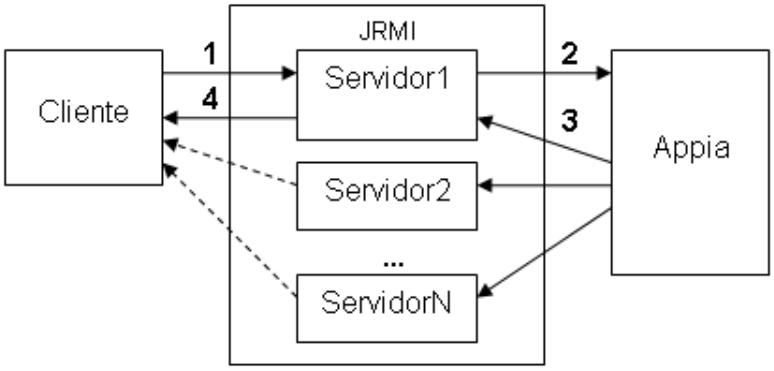

Figura 3.3: Diagrama de passagem de mensagens

Pode ainda ocorrer um cenário em que a réplica falha, mas o pedido já foi difundido correctamente para todas as réplicas correctas. No JRMI o cliente ao presenciar esta situação, não ficaria com dúvidas em assumir que foi um erro ocorrido no servidor. No FTRMI, o cliente de uma forma totalmente transparente nem se apercebe que o servidor falhou devido ao facto de receber a resposta duplicada enviada por todas as outras as réplicas. Quando qualquer réplica recebe o pedido vindo de ordem total uniforme, pode por si fabricar um pacote TCP exactamente igual ao que é preparado pela réplica que atendeu o pedido do cliente, incluindo o cabeçalho IP, TCP e JRMI. O pacote fabricado, está representado na fig. 3.3 a tracejado e utiliza a informação adicional necessária que

foi colocada no pedido distribuído pelo Appia, o IP origem, o IP destino, a porta origem, a porta destino, e os números de sequência (ACK e SEQ). A réplica que originalmente atendeu o cliente, pode responder normalmente pela sua conexão TCP e com as bibliotecas originais do JRMI, e todas as outras enviam um pacote fabricado. Com esta solução, o único caso de falha é aquele em que o servidor que estava directamente a atender o pedido falhe na terceira etapa, e todos os pacotes fabricados por todas as outras réplicas se perderem no caminho até ao cliente, não existindo qualquer tentativa de reenvio. Note-se que a probabilidade de isto acontecer é inversamente proporcional à quantidade de réplicas presentes.

Na conexão entre o cliente e o servidor está a ser utilizado o JRMI, logo o protocolo utilizado na camada de transporte da pilha TCP/IP é o TCP. Uma das características do protocolo TCP é a garantia de entrega de mensagens. Uma das formas de implementar esta característica é retransmitir mensagens que possivelmente foram perdidas ao longo da comunicação, através de um tempo limite para a confirmação de recepção das mesmas. Para este mecanismo funcionar correctamente, é necessário ignorar mensagens repetidas, para o caso em que cheguem duas retransmissões da mesma mensagem. Em resposta a um pedido efectuado pelo cliente, todos os servidores respondem com uma resposta idêntica e é interpretada como duplicada pelo protocolo TCP. O pacote fabricado descrito acima torna-se totalmente transparente para o cliente devido ao facto de uma conexão TCP rejeitar mensagens que já foram recebidas, ou seja, ao serem retransmitidas várias vezes a mesma mensagem apenas a primeira a chegar é aceite, as restantes são descartadas. Ao utilizarmos este mecanismo do protocolo TCP também pode causar uma perda de desempenho. Outra das características do protocolo TCP é o controlo de congestão. Este mecanismo permite ao TCP detectar quando é que a rede entre os dois interlocutores de uma ligação está congestionada, e os sintomas para esta detecção são calculados através do número de pacotes repetidos recebidos num curto espaço de tempo, indicando que alguns pacotes não foram dados como recebidos a tempo e pode haver congestionamento. Quando este mecanismo é accionado, o TCP baixa o ritmo de envio de mensagens, em número e tamanho. Desta forma, quando os pacotes fabricados forem recebidos pelo cliente, a ligação para o servidor pode sofrer uma baixa de desempenho devido ao TCP estar a reagir erroneamente aos pacotes recebidos em duplicado.

Capítulo 4

Implementação

A plataforma Fault-Tolerante Remote Method Invocation (FTRMI) foi implementada em Java e recorreu ao Appia para a concretização do conceito de máquina de estados distribuída. O FTRMI é um sistema distribuído que está dividido em três componentes principais, o cliente, o servidor, e o registo. O cliente e o registo mantiveram-se intactos perante os que são disponibilizados na plataforma Java Remote Method Invocation (JRMI) [31]. Não são necessárias quaisquer alterações a esses componentes, tanto a nível de implementação, como a nível de configuração. No servidor foram realizadas alterações a vários níveis, nomeadamente, a obtenção de identificadores para objectos, as políticas de multitarefa e a concretização da troca de estado entre os objectos.

O servidor tem ao seu dispor vários comandos que pode executar sobre o registo, sendo o mais relevante, registar um objecto remoto. O registo de um objecto recebe dois parâmetros, o nome a associar à instância do objecto que se está a disponibilizar e a respectiva instância. Uma parte das alterações realizadas no lado do servidor incidiram sobre esta operação de registo, de forma a tornar possível anunciar os objectos na plataforma FTRMI. O cliente utiliza a operação que permite procurar objectos no registo e obtém uma referência para onde o objecto está localizado.

4.1 Alterações no Servidor

4.1.1 Configuração do Carregador de Classes

A metodologia utilizada para implantar o FTRMI num sistema distribuído JRMI sem ser necessária recompilação de código, foi a substituição de uma classe da distribuição do JRMI por outra, que respeita a mesma interface. A substituição das classes ocorre no momento de início da aplicação através da modificação de argumentos do carregador de classes. O carregador de classes de uma máquina virtual Java realiza em tempo de execução a pesquisa das classes que vão sendo necessárias carregar para memória, ao longo da execução de um programa. Alterando a ordem pela qual o carregador de classes faz a pesquisa de classes, possibilita a substituição de uma classe originalmente disponi-

bilizada pelo Java, por uma classe modificada, mas que partilhe a mesma interface.

```
-Xbootclasspath/p:exemplo_jar.jar:.
```

Este argumento, colocado nos parâmetros de configuração de uma máquina virtual, permite que durante o seu arranque o carregamento de classes seja efectuado pela ordem inversa. Isto é, em primeiro lugar são carregadas as classes que se encontram em $exemplo\_jar.jar$, posteriormente, são carregadas as classes disponibilizadas na directiva corrente e, finalmente, as disponibilizadas pela distribuição do Java por omissão. Desta forma, é possível alterar qualquer classe que seja utilizada por um servidor JRMI. A classe escolhida foi a $UnicastServerRef$ que se responsabiliza na pilha de comunicação do JRMI (ver fig. 2.3) pela ligação entre as bibliotecas padrão estáticas, e as camadas dependentes de cada aplicação. A substituição desta classe permite interceptar todas as invocações executadas num servidor para todos os seus objectos remotos, sem interferir com a geração do $skeleton$, nem com as restantes camadas. A camada adicionada à pilha de comunicação do JRMI está sombreada na Fig. 3.1. A adição desta nova classe permite a utilização de um Sistema de Comunicação em Grupo (GCS) 2.2, também representado na figura, de uma forma totalmente transparente.

4.1.2 Configuração do Appia

O Appia é o GCS utilizado pelo FTRMI, que oferece diferentes características de comunicação, através da criação de canais. O Appia permite criar múltiplos canais de comunicação através da composição em camadas de diferentes tipos de serviço. O FTRMI utiliza um canal criado na plataforma Appia para disponibilizar três requisitos fundamentais: sincronia virtual, ordem total uniforme e noção de vista primária.

A sincronia virtual define um conceito de grupo de processos denominado de vista. Considera-se a mudança de vista quando algum dos processos entra ou sai deste grupo, voluntariamente ou falhando. A sincronia virtual garante que mensagens entregues numa vista foram enviadas precisamente na mesma vista, permitindo desta forma criar um ponto de sincronização entre os vários processos. A ordem total uniforme garante que a entrega de mensagens é efectuada pela mesma ordem, a todos os processos do grupo, sejam faltosos ou correctos. A noção de vista primária requer que no arranque um dos processos, e apenas um, seja definido como primário. Sendo assim, quando esse processo se une a outro e o informa que tem a vista primária, esta vista torna-se primária. A substituição do processo primário pode ocorrer durante a execução. A noção de vista primária tem a função de permitir que o sistema funcione em ambientes onde é possível a existência de partições, como por exemplo a Internet.

Uma partição ocorre quando um grupo de vários processos se separa em grupos distintos por falhas na rede. Este problema pode causar falhas de coerência ao FTRMI, por exemplo, as réplicas que constituem o sistema separam-se em dois grupos. Se existirem

clientes ligados às diferentes partições, ambas as partições podem divergir o seu estado em diferentes direcções. A noção de partições do Appia permite decidir de uma forma determinística que apenas uma das partições fique com o direito de processar pedidos. A restante partição fica em modo de espera até se conseguir ligar a uma partição primária, de forma a conseguir processar pedidos.

4.1.3 Obtenção de um Identificador Único

Cada objecto remoto, criado pelo JRMI, tem um identificador único (ObjID). Este identificador está presente na referência do objecto e é dependente da localização do servidor porque contém informação sobre a máquina virtual Java e da máquina física. Os identificadores não dão suporte a mecanismos de replicação. Um identificador de um objecto é, propositadamente, único e definido pelo servidor onde o objecto é instanciado, por forma a ser distinguido univocamente de todas as instâncias de todos os objectos. Este identificador é obtido pelo cliente através do registo. O identificador está presente em todas as mensagens trocadas entre o cliente e o servidor, de modo a distinguir qual a instância do objecto onde estão a ser executados os pedidos. Se os pedidos e respostas contêm este identificador na versão JRMI, então torna-se necessário ao FTRMI saber gerá-lo e interpretá-lo para manter a transparência total, de forma a disponibilizar o mesmo serviço. Assim, para cada instância, de cada objecto, é necessário encontrar um identificador único partilhado por todas as réplicas FTRMI.

O mecanismo utilizado para resolver este problema foi a passagem da responsabilidade de criação dos identificadores dos objectos do JRMI para o FTRMI. O algoritmo utilizado para calcular o identificador único, inicia-se com a criação de um $hash$ baseado no nome canónico da classe que vai dar origem à instância. É adicionada uma entrada numa tabela de dispersão que associa o identificador criado, representado pelo $hash$, à instância do objecto criado. As colisões de $hashs$ são resolvidas adicionando unidades ao valor resultante. A tabela de dispersão permite ainda, guardar informações independentes de cada objecto, como por exemplo, a referência para o fio de execução ($thread$) associado a cada. O algoritmo de atribuição de identificadores aos objectos criados, depende da sequência única de criação de todos os objectos em todas as réplicas, o que se espera que ocorra na maior parte das concretizações de servidores, uma vez que é a sequência usual. Quando os servidores FTRMI iniciam a sua tarefa, começam por instanciar todos os objectos que desejam disponibilizar, seguindo o modelo do JRMI. Ao serem instanciados existe um ponto em comum com todos os servidores, a ordem pela qual são instanciados, permitindo uma sincronização determinista com o GCS. Desta forma, é evitado o cenário em que um servidor recebe pedidos num determinado objecto, para o qual ainda não possui uma instância criada.

A Fig. 4.1 apresenta um cenário possível de cálculo de identificadores que são atribuídos aos objectos, demonstrando que o algoritmo é determinístico. Assumindo que existem

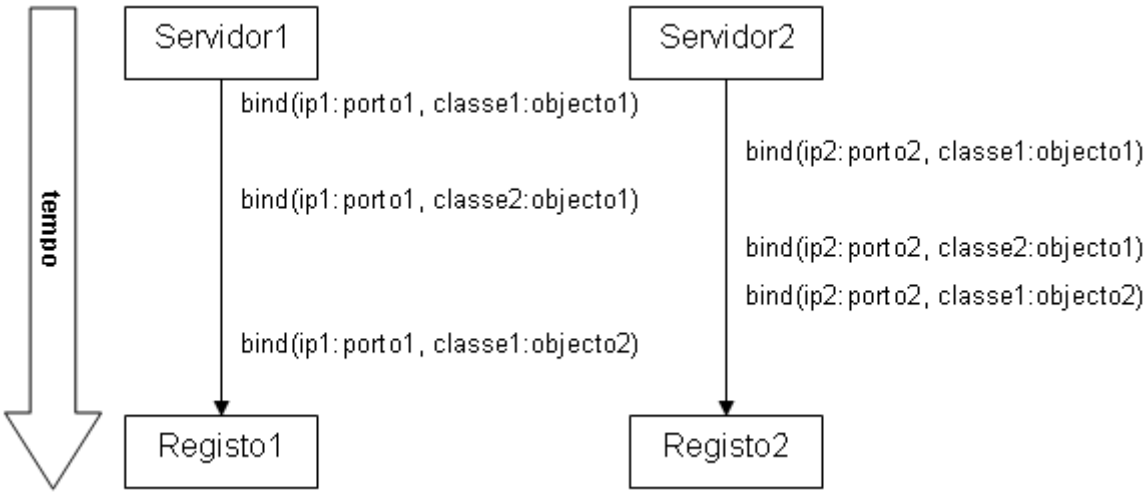

Figura 4.1: Cenário de criação de identificadores

duas classes diferentes que possuem a mesma $hash$, para o respectivo nome canónico, uma delas tem duas instâncias e, a outra, apenas uma instância. Inicialmente, o servidor1 regista o objecto1 da classe1 atribuindo-lhe N. Concorrentemente, o servidor2, seguindo a mesma ordem de criação como referido no algoritmo, regista o objecto1 da classe1 atribuindo-lhe o mesmo identificador N. À medida que o servidor1 avança, prepara-se para registar o objecto1 da classe2 e atribuí $N + 1$, visto que a classe2 possui a mesma $hash$ que a classe1. De seguida, o servidor2 regista o objecto1 da classe2, e o objecto2 da classe1 respectivamente, com os identificadores $N + 1$ e $N + 2$. Independentemente da classe que representa o objecto para o qual estão a ser criadas instâncias, as colisões são resolvidas incrementando unidades ao identificador inicial calculado para essa classe. Por fim, o servidor1 regista o último objecto obtém o identificador $N+2$. A possibilidade de todos os servidores calcularem os identificadores apenas com informação localmente disponível torna o algoritmo eficiente e determinista.

É possível garantir que todas as réplicas partilham o mesmo identificador para a mesma instância de cada objecto. O identificador foi atribuído na fase de instanciação de um objecto e por consequência o objecto foi registado no serviço de nomes com este identificador. Quando um cliente utiliza o servidor de nomes para encontrar a instância do objecto para qual vai prosseguir com a invocação, obtém o mesmo identificador. Quando um pedido JRMI é interceptado, o identificador que está presente no pedido foi devidamente atribuído pelo FTRMI e, portanto, é idêntico entre todas as réplicas permitindo a identificação da mesma instância em todas elas.

4.1.4 Camada FTRMI

A classe que substitui a classe $UnicastServerRef$ presente no JRMI, intercepta todos os pedidos realizados pelos clientes num servidor JRMI. Quando um pedido é recebido pela nova versão da classe é serializado e encaminhado para o GCS por forma a concretizar a replicação. O processamento do pedido é realizado em todos os servidores após a recepção da mensagem ordenada totalmente.

Em simultâneo, existe um mecanismo que escuta o tráfego transmitido pela camada TCP/IP com recurso à utilização da biblioteca libpcap [1]. A biblioteca libpcap foi criada com o objectivo de facilitar a criação e a captura de tráfego de rede. É desenvolvida em C/C++ e foi concretizada de forma a ser portável entre sistemas operativos. A ponte entre o Java e as funções utilizadas do libpcap foi realizada recorrendo ao Java Native Interface (JNI) [2]. A utilização da JNI ao invés de uma aplicação tradicional, permite a separação entre o código realizado em Java e a biblioteca libpcap. A separação facilita a portabilidade do FTRMI, uma vez que como Java é multiplataforma, a utilização do FTRMI fica limitada à disponibilidade do libpcap para outros sistemas operativos. Apesar disso, o libpcap foi concretizado para que possa ser compilado para diferentes sistemas operativos. Contudo, a execução de programas que usam a biblioteca libpcap para capturar e gerar tráfego de rede necessitam de privilégios de administrador.

A ligação entre a biblioteca compilada e o Java é automaticamente realizada pelo JNI, sendo necessário colocar a biblioteca compilada num local conhecido pela máquina virtual Java, ou directamente especificando os argumentos necessários. A integração do JNI com o libpcap permite que a camada TCP/IP seja escutada e devidamente interpretada, sendo possível a criação de um histórico de todas as ligações recebidas pelo servidor. Quando o JRMI entra em funcionamento e a classe substituída do FTRMI é invocada, também é inicializado um $daemon$ que se responsabiliza pela configuração e manutenção do libpcap e respectivos recursos necessários. A função principal do $daemon$ é manter em memória os números de sequência (ACK e SEQ) para cada quarteto representativo de uma conexão TCP.

Ao escutarmos a camada TCP/IP torna esta aproximação pouco eficiente e muito invasiva, uma vez que tem acesso total ao tráfego do servidor. Contudo, para melhorar esta aproximação, o libpcap disponibiliza um mecanismo que permite configurar o tipo de tráfego a capturar, e no âmbito do FTRMI o libpcap foi limitado para apenas capturar tráfego TCP. Contudo, quando um cliente JRMI inicia um pedido de invocação remota a um servidor, existem várias etapas de comunicação, mesmo antes de o pedido ser efectivamente realizado. Esta particularidade permite que o filtro efectuado sobre o tráfego capturado seja melhorado.

A Fig. 4.2 representa em pormenor, a troca de mensagens entre o cliente e o servidor

[1] http://www.tcpdump.org/
[2] http://java.sun.com/docs/books/jni/

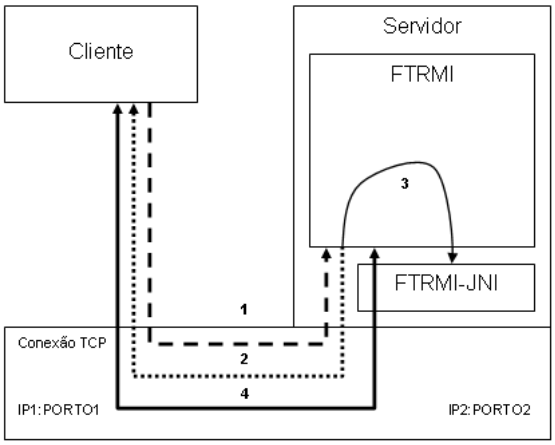

Figura 4.2: Diagrama de troca de mensagens entre cliente e servidor

em JRMI. Em primeira instância, o cliente inicia a comunicação com o servidor efectuando uma troca de mensagens. Assim, o universo de pedidos aos quais o $daemon$ deve capturar, pode ser reduzido analisando a origem das conexões. No caso do JRMI, o cliente inicia a conexão, logo o $daemon$ deve ignorar todas as conexões iniciadas por ele próprio. Nesta etapa, o FTRMI tem em sua posse o pedido do cliente, bem como o quarteto que representa a conexão (IP e porto do cliente, IP e porto do servidor) pela qual o pedido foi enviado. Entre a segunda e a terceira etapa, o FTRMI consulta o $daemon$ para o quarteto da respectiva conexão, obtendo os respectivos números de sequência (ACK e SEQ), e marca a respectiva ligação como em uso. Este mecanismo permite ao $daemon$ descartar informação irrelevante para a plataforma FTRMI, utilizando o tempo como critério de selecção, uma vez que existe a percepção de quando é que uma conexão está a ser utilizada.

Na segunda etapa, o servidor responde ao cliente que o pedido foi efectuado com sucesso. O sincronismo entre o FTRMI e o $daemon$, é crucial para o funcionamento correcto deste mecanismo, uma vez que a resposta gerada pelo JRMI nesta etapa altera os números de sequência (ACK e SEQ) que o $daemon$ possui. Assim sendo, o pedido que vai ser serializado e colocado no GCS só pode ser libertado após a confirmação enviada para o cliente. Na terceira etapa, após os valores de sequência já estarem devidamente actualizados, a difusão do pedido prossegue normalmente para todos os servidores recorrendo ao GCS. Finalmente, o pedido é processado por todos eles, e a resposta é enviada ao cliente.

4.1.5 Troca de Estado

A plataforma FTRMI necessita de um algoritmo de troca de estado entre os objectos, para permitir a entrada de novas réplicas no sistema. A entrada e saída de réplicas é um acontecimento provável e manuseado de forma a permitir a continuidade do sistema. O

Appia é configurado pelo FTRMI para disponibilizar um canal de comunicação para um grupo de processos com sincronia virtual, que fornece pontos de sincronização quando há alterações no grupo, e ordem total uniforme, que garante a ordem de entrega de mensagens. Quando um sistema distribuído que utilize o FTRMI está em pleno e uma nova réplica entra no respectivo grupo, o ponto de sincronização que o Appia oferece, é utilizado para a troca de estado de todas as instâncias dos objectos. A troca de estados requer que todos os objectos partilhados sejam serializáveis, contudo, esta funcionalidade é fornecida pela máquina virtual Java se os objectos forem assinalados com a interface $Serializable$. A serialização de objectos, permite o envio de um objecto com todo o conteúdo presente na instância, para todas as réplicas.

Utilizando o ponto de sincronização fornecido pelo GCS, a troca de estado torna-se num mecanismo de fácil implementação. Quando existe a entrada ou saída de algum processo do grupo, o GCS informa através de uma mensagem especial que é necessário alterar a vista do grupo. O FTRMI reage à notificação de alteração da vista do grupo, parando temporariamente o reencaminhamento de pedidos. A paragem de reencaminhamento de pedidos, permite ao GCS determinar quais as mensagens a entregar, antes de informar todos os processos que a vista foi alterada. Quando os processos são notificados em como se encontram numa nova vista, podem considerar a existência de um ponto de sincronismo, porque todos os processos sabem exactamente quais os processos que saíram e/ou entraram. Um algoritmo simples para sincronizar o estado entre todas as réplicas FTRMI, seria o envio do estado de todos os objectos, após a notificação de entrada numa nova vista. Assim, seria possível em todas as réplicas determinar qual o último estado para cada objecto. Contudo, esta abordagem sofre problemas de desempenho, assumindo que o número de objectos a sincronizar pode escalar.

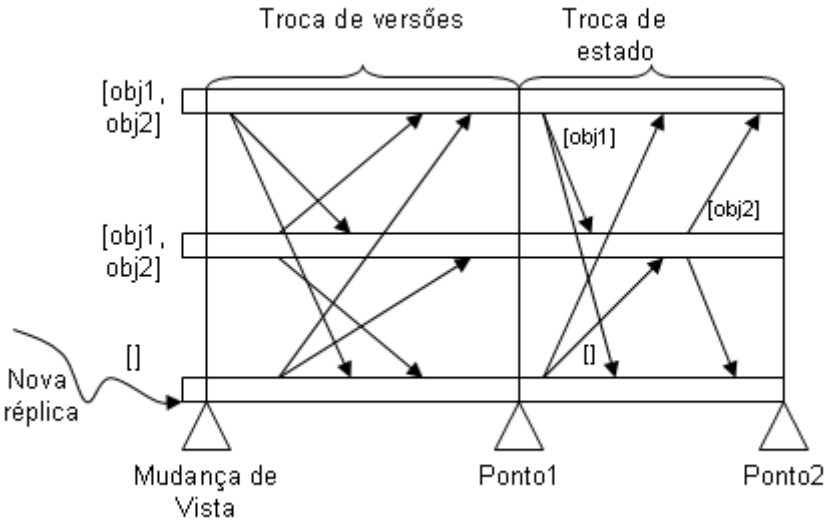

Figura 4.3: Algoritmo de troca de estado

A Fig. 4.3 retrata um algoritmo optimizado de sincronização de estado entre réplicas FTRMI. Após a mudança de vista, as réplicas efectuam um algoritmo de troca de estado em duas rondas. Durante a primeira ronda, todas as réplicas difundem em ordem total uniforme, a versão dos estados de todos os objectos que possuem. A versão é constituída por três números, e tem como único objectivo comparar a actualidade do estado de cada réplica. Os três números que a constituem são, número de sequência, número de vista, e um $hash$. O número de sequência é incrementado a cada invocação realizada na instância do objecto, o número de vista é um número fornecido pelo GCS que identifica univocamente cada vista, e o $hash$ é determinado pelo Java para cada objecto.

Quando todas as réplicas recebem esta informação, de uma forma determinista, podem decidir quais as réplicas que têm o estado mais recente. Após todas réplicas terem anunciado as versões dos estados de cada objecto, entram na segunda ronda. A segunda ronda é onde são efectuadas as trocas de estados por parte de todas as réplicas, note-se que para algumas réplicas pode não ser necessário enviar estado. Quando todos os estados chegam a todas as réplicas são imediatamente substituídos garantindo que todas elas se encontram no mesmo estado após um ponto de sincronização. Este mecanismo, considerando um sistema que partilha dois objectos distintos, permite que sejam réplicas distintas a enviar o estado de cada objecto. Ao terminar a segunda ronda, todas as réplicas contêm o último estado de todos os objectos e finalmente podem difundir os pedidos que chegaram durante a execução do algoritmo.

No JRMI está implementado um Distributed Garbage Collector (DGC) que serve exclusivamente para libertar a memória associada a objectos distribuídos que já não estão a ser utilizados. No caso do FTRMI, como utiliza as bibliotecas do JRMI para comunicação entre os clientes e os servidores, também possui este mecanismo por defeito. Contudo, para garantir que o FTRMI disponibiliza um objecto de forma ininterrupta, é necessário desactivar o mecanismo de DGC oferecido por defeito pelo JRMI. A desactivação deste mecanismo ocorre no momento de criação de objectos remotos, onde é possível definir que estes objectos têm carácter permanente, evitando que o DGC entre em funcionamento.

4.1.6 Controlo de Concorrência

A plataforma JRMI utiliza um modelo multitarefa para processar de forma paralela e eficiente os pedidos vindos dos clientes. Por outro lado, o Appia, que é a plataforma que faz a comunicação entre todas as réplicas, utiliza um modelo mono tarefa. Sendo assim, é necessário encontrar um sincronismo entre as duas plataformas, para a garantir o determinismo no processamento de pedidos, individualmente em cada réplica. A ligação entre estes dois modelos é crucial para o máximo desempenho do FTRMI.

O JRMI assume, que se for necessário controlo de acesso a algum recurso disponível num objecto remoto, é da responsabilidade do programador garanti-lo. No FTRMI deve ser possível descriminar no tempo a sequência de pedidos processados em cada objecto

remoto distinto. Desta forma, podemos garantir que a ordem de todos os pedidos é igual em todos os servidores e ainda que se encontram no mesmo estado se tais operações forem deterministas. Os pedidos ordenados totalmente chegam ao FTRMI através de um canal de comunicação disponibilizado pelo Appia, e são posteriormente invocados nos objectos.

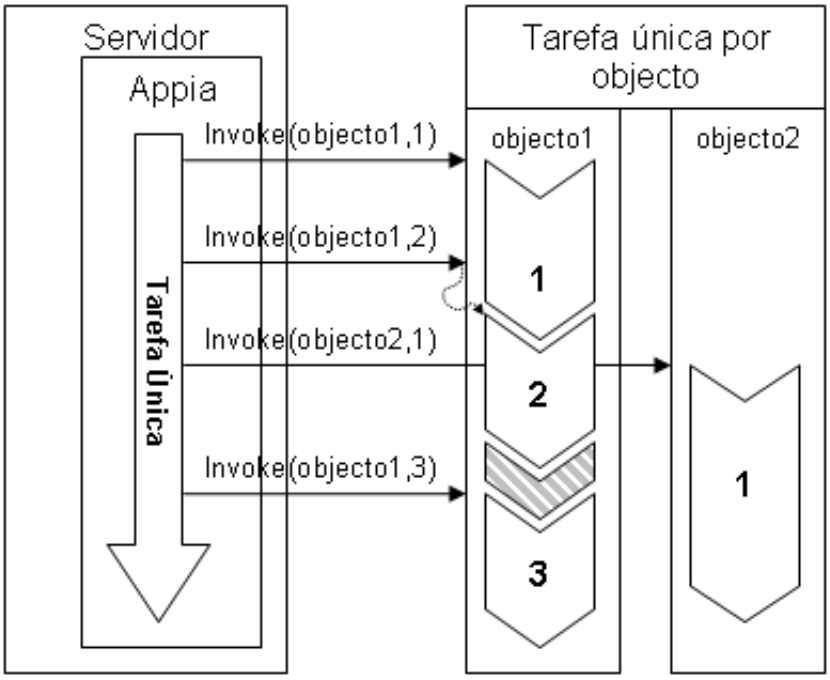

Figura 4.4: Modelo de tarefas

A Fig. 4.4 mostra como os pedidos são processados de forma sequencial no FTRMI, garantindo que não existem problemas de concorrência com os recursos contidos num objecto remoto. Cada objecto remoto possui uma fila e uma tarefa associada para garantir a eficiência e independência entre objectos. A fila de espera, serve para acumular múltiplos pedidos vindos do canal de comunicação utilizado entre os servidores, e a tarefa única para impor a ordem de sequência de processamento.

À medida que os pedidos vão chegando, são colocados na fila de espera associada à tarefa de cada objecto. Quando existe uma mudança de vista e um objecto tem a necessidade de enviar a última versão do seu estado, é fácil garantir que não existem pedidos na fila de espera, mas em contrapartida não é possível garantir que não está nenhum pedido em execução. Quando os processos são notificados da entrada numa nova vista, os processos têm a garantia em como não vão chegar pedidos de clientes devido ao algoritmo da troca de estado. Então, para resolver este problema, é possível colocar na fila de tarefas a executar uma tarefa vazia. Quando o resultado da tarefa vazia, previamente inserida, for retornada obtemos a garantia de como não existem pedidos pendentes na fila, e não existe nenhum pedido a ser processado. Posteriormente, a versão pode ser obtida e enviada para as restantes réplicas.

4.2 Tolerância a Faltas

4.2.1 Resposta Múltipla

No FTRMI, os clientes só comunicam directamente com um dos servidores, por forma a manter a compatibilidade com a arquitectura JRMI. Se um pedido é processado pelo FTRMI, então foi processado por todos os servidores, mas apenas o servidor que originalmente recebeu o pedido responde de volta ao cliente no protocolo JRMI. Como o cliente só efectua a comunicação directa com um dos servidores, no caso de este falhar, ficaria sem resposta. Por outro lado, não iria saber como interpretar as respostas vindas de outros servidores sem estar devidamente adaptado para isso. Uma alternativa simples para adaptar os clientes de uma forma totalmente transparente para receberem múltiplas respostas sem qualquer alteração, seria descartando respostas recebidas em duplicado. Para este efeito, seria necessário que as respostas fossem totalmente iguais. Ao analisar o protocolo TCP podemos observar que isto já é posto em prática. Quando existe uma conexão TCP, para garantir a entrega de mensagens, pode ser necessário repetir o envio de algumas mensagens que caso cheguem em duplicado, são ignoradas. A implementação FTRMI pode tirar partido deste mecanismo para ignorar as respostas com origem em diferentes servidores.

O FTRMI, quando recebe um pedido vindo de um cliente através de uma conexão TCP, deve reencaminhar o pedido para o canal de comunicação fornecido pelo GCS. Após o pedido ser devidamente ordenado e entregue a todos os servidores, incluindo ele próprio, este vai posteriormente ser executado. O pedido é processado e, em simultâneo, é preparada uma resposta que vai ser enviada com o resultado do processamento. Note-se que o pedido pode demorar alguns segundos a ser atendido, e nem todos os servidores demoram o mesmo tempo a processá-lo. Quando um pedido é inicialmente difundido para todos os servidores, leva a informação necessária para que todos eles consigam gerar uma resposta para o cliente. A informação necessária é recolhida com recurso à biblioteca libpcap, e é constituída pelo, IP do destinatário, IP do remetente, o porto do destinatário, o porto do remetente, e os respectivos números de sequência (SEQ) e confirmação (ACK).

Quando um servidor acaba de processar um pedido verifica se o cliente está ligado directamente a ele, se sim responde-lhe directamente pela conexão JRMI existente. Caso contrário, com fundamento na informação extra, recebida no pedido, gera manualmente uma mensagem de resposta, construindo um pacote ao nível TCP/IP e preenchendo os cabeçalhos necessários para que este chegue correctamente ao cliente. O pacote contém como destinatário o respectivo cliente remetente e os respectivos portos de acesso. Os números de sequência e confirmação devem ser trocados um pelo outro, e devidamente incrementados com o tamanho da resposta a ser enviada. Ainda assim, é necessário recalcular os $checksums$ realizados pelas camadas TCP e IP, visto que a única forma de colocar este pacote na rede, é colocá-lo exactamente como queremos que ele seja entre-

gue. A geração do pacote que contém a resposta, deve ser realizada com exactidão, uma vez que é colocado na rede exactamente igual aquando foi gerado.

O cliente recebe as respostas de todos os servidores, sendo o TCP o responsável por descartar as respostas repetidas. As respostas foram todas geradas individualmente em cada servidor, contudo, foi utilizado um mecanismo determinístico. A utilização de um mecanismo determinístico, faz com que todas as respostas sejam totalmente iguais, incluindo a que foi gerada pelo mecanismo do JRMI.

Capítulo 5

Avaliação

A avaliação compara o desempenho entre duas plataformas, a Fault-Tolerante Remote Method Invocation (FTRMI) e a Autonomous Replication Management (ARM). O ARM foi seleccionado para esta comparação devido à partilha da mesma linguagem de programação, do modelo da plataforma de suporte à distribuição e modelo de replicação. Contudo, estas plataformas utilizaram diferentes abordagens à transparência e optaram por Sistemas de Comunicação em Grupo (GCS) 2.2 distintos, respectivamente o Appia [15] e o Jgroup [17]. Adicionalmente, foram realizados testes de controlo utilizando a plataforma Java Remote Method Invocation (JRMI) [31] originalmente disponibilizada pelo Java. O desempenho foi comparado entre as várias plataformas utilizando duas métricas. A latência, que mede o tempo médio de cada invocação desde o momento em que o pedido é realizado até ao momento em que a resposta é recebida. A segunda métrica, a quantidade de tráfego, soma todos os octetos transferidos pela rede durante cada invocação. O tráfego foi estimado utilizando a informação disponibilizada pelo sistema operativo relativamente aos interfaces de rede. Embora estes resultados não sejam exactos, a execução dos testes em condições rigorosamente idênticas sugere que a percentagem de erro deverá ser semelhante em todos os testes efectuados. Finalmente, foi realizado um teste de estabilidade da plataforma desenvolvida. Este teste, teve a duração de aproximadamente 24 horas, durante as quais se provocou aleatoriamente erros ao nível do componente de servidor, testando também a capacidade de regeneração da plataforma.

Os testes foram realizados utilizando uma rede Ethernet a 100Mb/s comutada representada na Fig. 5.1. Todos os componentes ligados a esta rede serviram apenas para executar os testes às aplicações, não produzindo qualquer outra actividade. Durante os testes, quatro servidores dedicados foram utilizados como réplicas. Estes servidores continham entre 2Gb e 8Gb de memória volátil e processadores AMD Opteron™248, Intel®Xeon®CPUs 3060 e X3370. Os pedidos foram efectuados por quatro máquinas cliente com processadores Intel®Core™2 Duo CPUs E7500 e E8300 com 2Gb de memória volátil cada. Todos os anfitriões utilizaram um sistema operativo Linux v.2.6 e Java SE Runtime 1.6.0_26.

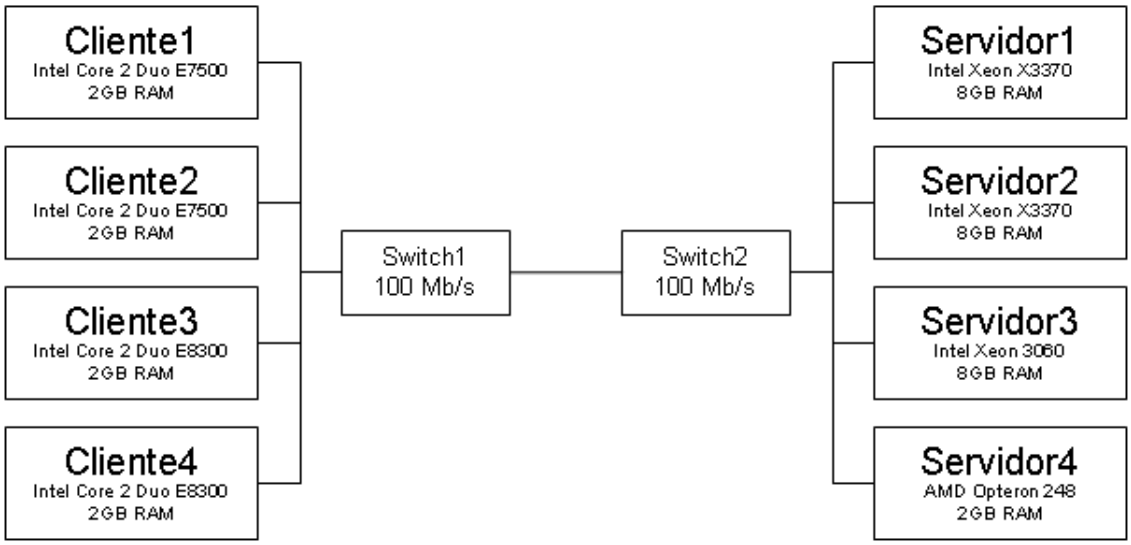

Figura 5.1: Mapa da rede de testes

5.1 Teste de Resiliência

O teste de resiliência à plataforma FTRMI teve a duração de 24 horas. Neste teste, o código do FTRMI foi instrumentado para que durante a execução normal da plataforma, ocorressem falhas propositadas em pontos chave do processo de tratamento de pedidos de forma a testar se todos os componentes do sistema reagiam bem às falhas causadas.

Os pontos de falha criados estão representadas na Fig. 5.2. A primeira falha ocorre antes de o pedido chegar ao FTRMI pelo que se espera uma resposta padrão do JRMI e que consiste numa excepção Java. A segunda falha ocorre antes de o servidor que recebe o pedido conseguir transmitir o pedido para os outros servidores. O cliente é novamente notificado com uma excepção Java, e como o pedido não chegou a nenhum servidor não é processado. A terceira falha ocorre quando um servidor já recebeu um pedido que já foi ordenado totalmente entre todos os servidores, não processa o pedido e não gera uma resposta ao cliente. A quarta falha ocorre se o servidor que originalmente recebeu o pedido falha, contudo a resposta pode ser devidamente entregue por outro servidor. A quinta falha representa um servidor que estava a gerar uma resposta a um pedido recebido por outro servidor, contudo à semelhança da quarta falha a resposta pode ser devidamente entregue. No caso da terceira, quarta e quinta falhas, a resposta ao pedido pode não ser devidamente entregue ao cliente. Após o pedido ter sido entregue devidamente ordenado pelo GCS, todos os servidores o processam e em consequência geram uma resposta, contudo pode não ser devidamente entregue. Neste caso, o cliente tem possibilidades de receber a resposta ao pedido por intermédio de qualquer servidor, mesmo quando o servidor ao qual está directamente ligado falhe. Caso não receba resposta irá ser notificado com uma excepção Java quando a ligação inicial com o servidor terminar.

Neste teste de resiliência foi utilizada a invocação de um método remoto sem argu-

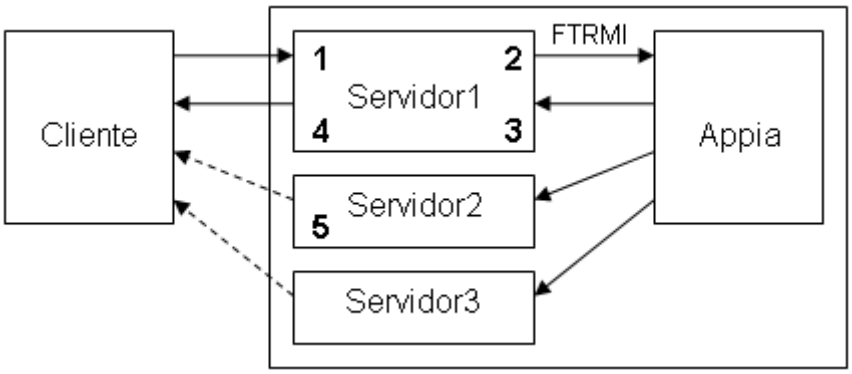

Figura 5.2: Cinco pontos de falha

mentos, e com o retorno de um inteiro longo representado na Fig. 5.3. As probabilidades de falha para os cincos pontos de falha foram regulados para 0.05 pontos percentuais. Neste teste, foram inicializados quatro servidores um em cada máquina disponível, e quatro clientes um em cada máquina disponível para os clientes, em que cada cliente ligou-se a um servidor distinto. O número de execuções durante as 24 horas foi de aproximadamente 2,5 milhões por cada cliente, o que perfaz o total de 10 milhões no total. O inteiro longo retornado pelo método utilizado foi nos locais dos pontos de falha escrito para ficheiro com alguma informação adicional relevante para manter um historial de toda a execução. Após a execução dos testes o histórico foi devidamente interpretado para detectar uma possível falha do sistema, e verificou-se que tudo aconteceu como previsto.

```
public long doComputeTest() {
   return increment++;
}
```

Figura 5.3: Método utilizado no teste de resiliência

5.2 Teste de Desempenho

Os testes de comparação entre o FTRMI e o ARM utilizaram um método remoto que tem como argumento uma cadeia de caracteres e retorna um inteiro que vai sendo incrementado a cada invocação, este método está representado na Fig. 5.4. No ARM, a replicação é visível para o cliente, que fica a fazer parte de um grupo de sincronia virtual para enviar o pedido e utiliza a primeira resposta recebida. Em contraste, o FTRMI esconde a replicação do cliente que fica fora do grupo de sincronia virtual mas também é obrigado a esperar pela primeira resposta que pode ser gerada pelo servidor ao qual endereçou o pedido ou qualquer outro servidor. A presença da cadeia de caracteres como argumento

é importante para testar o impacto da fragmentação no desempenho. Para tal, foram realizados testes utilizando cadeias de caracteres de 0 e de 2000 octetos. O inteiro que está a ser incrementado no método representa uma operação desprezável e serve apenas para garantir que no fim dos testes todos os pedidos foram executados, confiando que a ordem pela qual foram executados foi garantida pelas plataformas de comunicação em grupo.

```
public long doCompute(String str) {
   return increment++;
}
```

Figura 5.4: Método utilizado nos testes de desempenho

Foram experimentadas três configurações distintas do canal de comunicação Appia que suporta o FTRMI. Os resultados apresentados como FTRMI-1 representam a utilização de um algoritmo de ordem total que utiliza o conceito de coordenador. Nos resultados apresentados como FTRMI-2 foi utilizado um algoritmo de ordem total baseado em ordem causal. Nos resultados apresentados como FTRMI-3 foi utilizado um algoritmo de ordem total uniforme, que garante a entrega de mensagens a processos correctos ou faltosos. Todos os algoritmos são totalmente independentes da camada FTRMI, sendo disponibilizados no pacote padrão do Appia e podem ser seleccionados alterando a configuração do Appia. Contudo, o desempenho destes algoritmos pode variar com a topologia da rede, o número de participantes e a granularidade do tráfego. Os testes não consideraram a falha de processos, pois seria complicado replicar o mesmo cenário em vários testes distintos.

As medições foram iniciadas em todos os testes após uma primeira fase em que as máquinas virtuais Java são estimuladas, por forma a carregar para memória todas as classes necessárias à execução, e possibilitar o funcionamento do compilador $just-in-time$ (JIT). Para diminuir o impacto do factores externos nas medições, os resultados apresentados são a média de 500.000 invocações remotas realizadas por cada cliente.

A localização dos servidores que compõem o registo reflectem o desenho de cada plataforma. No FTRMI, o servidor de registo coexiste no mesmo anfitrião que o servidor de objectos e referencia apenas este, para N servidores de objectos existem N servidores de registo. O balanceamento de carga é assegurado quando todos os servidores possuem o mesmo número de clientes. No ARM, o servidor de registo replicado encontra-se no servidor1 de acordo com a Fig. 5.1 por ser aquele que apresenta mais poder computacional.

As colunas representadas nos gráficos são os resultados obtidos para um configuração possível entre número de servidores e número de clientes. Posto isto, s representa o número de servidores e c o número actual de clientes ligados a todos os servidores de forma homogénea.

5.2.1 Latência

Os resultados de avaliação de latência estão representados na Fig. 5.5. A comparação entre as colunas *s=2,c=4* e *s=4,c=4* mostra que a latência sofre grandes alterações quando o número de servidores aumenta, o que confirma a fraca escalabilidade dos algoritmos necessários para a concretização da máquina de estados distribuída.

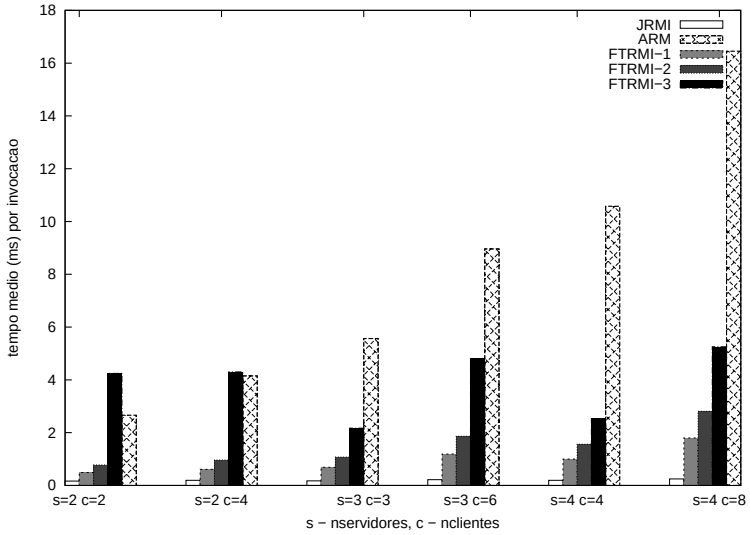

Figura 5.5: Média da latência entre invocações remotas sem argumentos

Como esperado, a plataforma JRMI tem um excelente desempenho e obtém os melhores resultados. O FTRMI-1 e o FTRMI-2, os dois algoritmos de ordem total regular, têm um desempenho semelhante existindo pequenas variações regulares. O FTRMI-3 apresenta latências demasiado altas quando a afluência de pedidos é reduzida, mas em contrapartida, quando existem muitos pedidos em simultâneo consegue reduzir as latências e em alguns casos aproxima-se do FTRMI-1 e FTRMI-2, devido à capacidade de agregação de pedidos. O ARM quando o número de servidores é 2 mostra bom desempenho, piorando substancialmente quando a quantidade de servidores aumenta.

Os resultados dos testes em que o argumento do método invocado foi preenchido com 2000 octetos aleatórios estão apresentados na Fig. 5.6. Todas as plataformas reagem da mesma forma aumentando regularmente os tempos obtidos, confirmando o factor de impacto da fragmentação. No FTRMI-3 os tempos aumentaram relativamente pouco, devido à capacidade de agrupamento de pedidos. Finalmente, é de notar o excelente desempenho da plataforma JRMI não replicada, que confirma o impacto negativo da replicação, ultrapassando mesmo os ganhos do balanceamento de carga.

5.2.2 Quantidade de Tráfego

Os resultados contribuem para justificar o tráfego gerado com a invocação do método sem argumentos observado nas Figs. 5.7 e 5.8. As elevadas quantidades de tráfego pro-

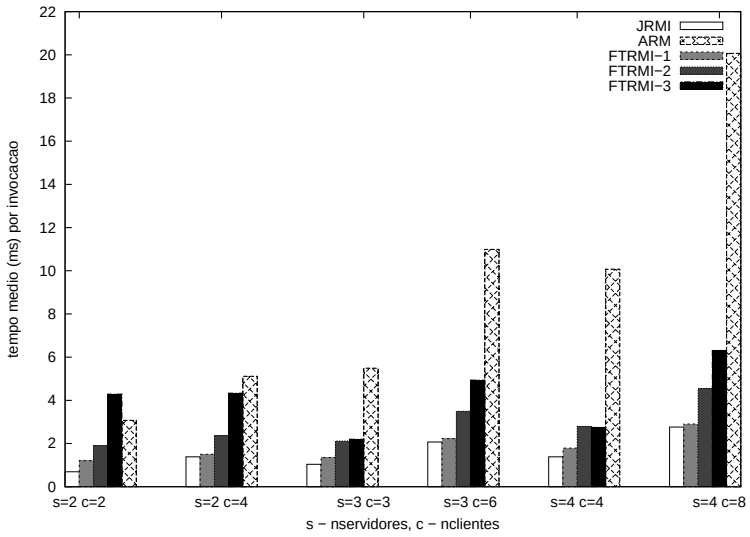

Figura 5.6: Média da latência entre invocações remotas com 2000 octetos de argumentos

duzidas pelo ARM explicam os valores da latência observados. Este valor é atribuído às características internas dos protocolos utilizados pelo ARM. Neste âmbito, importa referir que o cliente executa parte do protocolo de sincronia virtual e necessita por isso de trocar mensagens com todas as réplicas dos servidores. Em contraste, graças à transparência, o FTRMI utiliza o mesmo tráfego entre o cliente e o servidor que a plataforma JRMI, acrescentando o tráfego das respostas vindo de todos os servidores. Os resultados mostram que no ARM, o tratamento de uma invocação consome 5 a 10 vezes mais tráfego entre os servidores do que entre os clientes e os servidores. Uma comparação entre os dois algoritmos de ordenação total regular estudados para o FTRMI mostra que o FTRMI-2 consome sempre menos tráfego que o FTRMI-1, mas possuí um atraso mais elevado. Esta diferença está directamente ligada ao número de mensagens utilizados pelo protocolo de ordem total e respectivo tempo de envio. O FTRMI-3 utiliza um algoritmo de ordem total uniforme que em contraste com a ordem total regular garante que as respostas são processadas por todos os servidores independentemente se são correctos ou não, utilizando ligeiramente mais tráfego que os algoritmos genéricos. No ARM, outro factor de referência é o aspecto do tráfego decrescer quando a carga está no máximo, e ser ortogonal ao número de servidores.

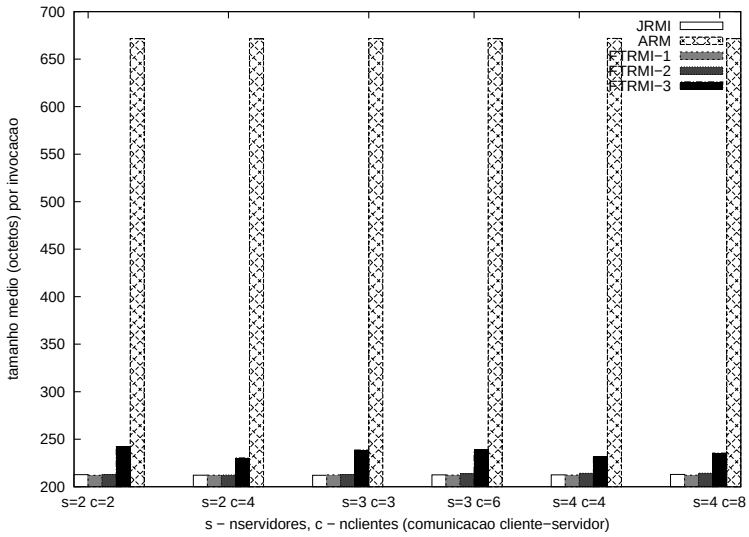

Figura 5.7: Tamanho médio para cada invocação, sem argumentos, entre 1 cliente e 1 servidor

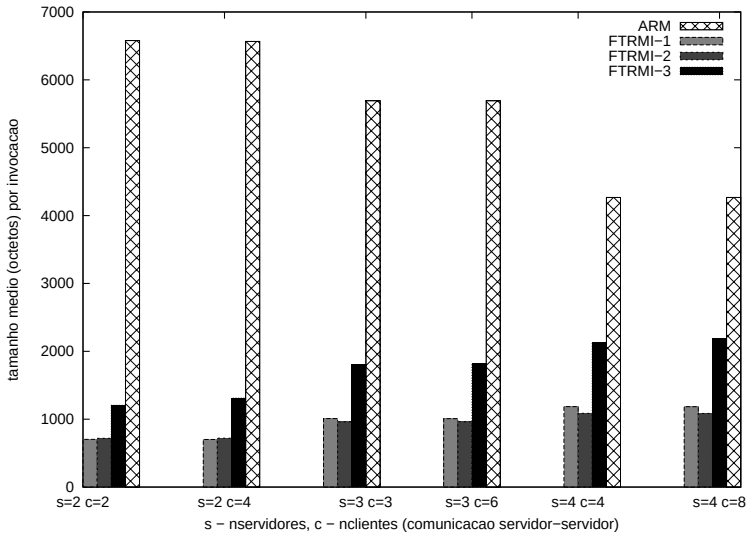

Figura 5.8: Tamanho médio para cada invocação, sem argumentos, entre cada servidor

Os resultados de tráfego para invocações remotas em que o argumento é preenchido com 2000 octetos aleatórios estão retratados nas Figs. 5.9 e 5.10. Estes resultados representam a generalidade do que foi observado nos testes com o argumento vazio reflectindo o aumento de tamanho das mensagens e respectiva fragmentação.

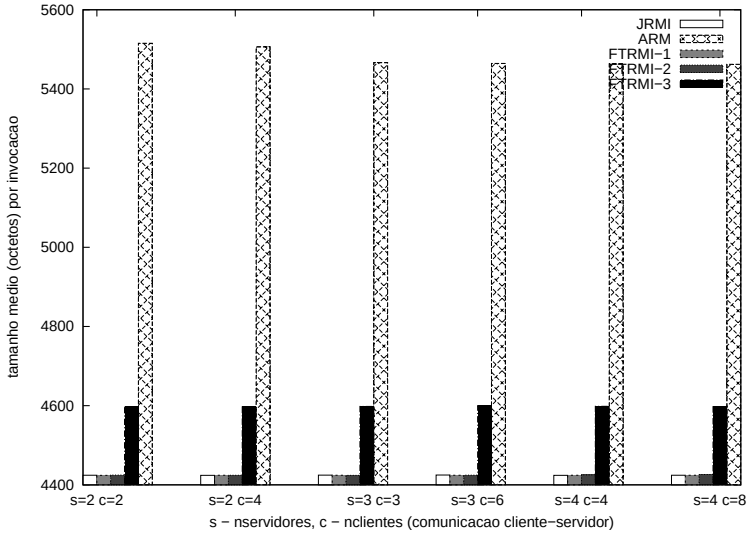

Figura 5.9: Tamanho médio para cada invocação, com 2000 octetos de argumento, entre 1 cliente e 1 servidor

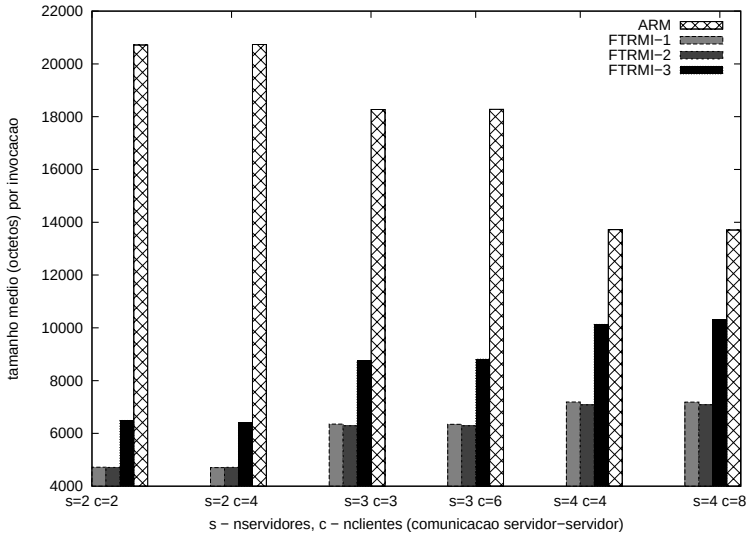

Figura 5.10: Tamanho médio para cada invocação, com 2000 octetos de argumento, entre cada servidor

Capítulo 6

Conclusão

A plataforma Fault-Tolerante Remote Method Invocation (FTRMI) descrita neste documento, consiste numa plataforma de replicação activa, totalmente transparente para aplicações que utilizem o Java Remote Method Invocation (JRMI) [31]. O FTRMI estende o paradigma de comunicação cliente/servidor disponibilizado pelo JRMI com uma camada adicional de comunicação no lado do servidor. A transparência que oferece permite, a uma qualquer aplicação JRMI actualmente desenvolvida ficar totalmente replicada sem qualquer recompilação, sendo apenas necessário executar o servidor com as bibliotecas disponibilizadas. A avaliação apresenta resultados que permitem ao FTRMI ser uma alternativa viável, por comparação com diferentes abordagens ao problema. Contudo, é necessário realizar mais testes que permitam avaliar como é que as plataformas reagem tendo em conta que o tamanho da resposta também provoca fragmentação. O FTRMI apresenta uma pequena, mas não desprezável penalização em relação a sistemas centralizados como JRMI original, que não oferecem quaisquer tolerância a faltas.

Abreviaturas

ACK	Acknowledgement Number
API	Application programming interface
ARM	Autonomous Replication Management
CORBA	Common Object Request Broker Architecture
DCOM	Distributed Component Object Model
DGC	Distributed Garbage Collector
DJOM	Distributed Java Object Model
DNS	Domain Name System
DOF	Distributed Object Framework
DR	Dependable Registry
EGMI	Internal Group Method Invocation
GCS	Group Communication System
IDL	Interface Description Language
IGMI	External Group Method Invocation
IOGR	Object Group Reference
IOR	Interoperable Object Reference
IP	Internet Protocol
IPC	Inter-process Communication
IRL	Interoperable Replication Logic
JERI	Jini Extensible Remote Invocation
JIT	Just-in-time Compilation
JNI	Java Native Interface
JOM	Java Object Model
JRMI	Java Remote Method Invocation
JRMP	Java Remote Method Protocol
OMG	Object Management Group
ORB	Object Request Broker
PGMS	Partitionable Group Membership Service

RM Replica Management
RMI Remote Method Invocation
ROG Remote Object Group
RPC Remote Procedure Call
RRO Replicated Remote Object

SEQ Sequence Number
SMS State Merging Service

TCP Transmission Control Protocol

Bibliografia

[1] Ken Arnold, Robert Scheifler, Jim Waldo, Bryan O'Sullivan, and Ann Wollrath. *Jini Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999.

[2] Henri E. Bal, Raoul Bhoedjang, Rutger Hofman, Ceriel Jacobs, Koen Langendoen, Tim Rühl, and M. Frans Kaashoek. Performance evaluation of the orca shared object system. *ACM Transactions on Computer Systems*, 16, 1998.

[3] Roberto Baldoni, Stefano Cimmino, Carlo Marchetti, and Alessandro Termini. Performance analysis of java group toolkits: A case study. In *Proceedings of the International Workshop on scientiFic engIneering of Distributed Java applIcations (FIDJI'2002)*, November 28–29 2002.

[4] Arash Baratloo, P. Emerald Chung, Yennun Huang, Sampath Rangarajan, and Shalini Yajnik. Filterfresh: hot replication of java RMI server objects. In *Proc. of 4th USENIX Conf. on Object-Oriented Technologies and Systems (COOTS'98)*, volume 4. USENIX Association, 1998.

[5] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. *SIGOPS Oper. Syst. Rev.*, 21:123–138, November 1987.

[6] K. Birman and R. van Renesse, editors. *Reliable Distributed Computing With the ISIS Toolkit*. IEEE CS Press, March 1994.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[8] Rachid Guerraoui and Luís Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[9] Leslie Lamport. On interprocess communication. *Distributed Computing*, 1:86–101, 1986. 10.1007/BF01786228.

[10] Jason Maassen, Thilo Kielmann, and Henri E. Bal. Efficient replicated method invocation in java. In *In ACM 2000 Java Grande Conference*, pages 88–96, 2000.

[11] Silvano Maffeis. The object group design pattern. Technical report, Cornell University, Ithaca, NY, USA, 1996.

[12] Carlo Marchetti, Massimo Mecella, Antonino Virgillito, and Roberto Baldoni. An interoperable replication logic for CORBA systems. In *Proceedings of the International Symposium on Distributed Objects and Applications - DOA'00*, Antwerp, Belgium, September21–23 2000. Institute of Electrical and Electronics Engineers.

[13] H. Meling and B. E. Helvik. ARM: Autonomous replication management in jgroup. In *Proc. of the 4th European Research Seminar on Advances in Distributed Systems (ERSADS'01)*, Bertinoro, Italy, May 2001.

[14] Hein Meling, Alberto Montresor, Bjarne E. Helvik, and Ozalp Babaoglu. Jgroup-arm: a distributed object group platform with autonomous replication management. *Softw. Pract. Exper.*, 38:885–923, July 2008.

[15] Hugo Miranda, Alexandre Pinto, and Luís Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of The 21st International Conference on Distributed Computing Systems (ICDCS-21)*, pages 707–710, Phoenix, Arizona, USA, April 16–19 2001. IEEE Computer Society.

[16] M.J. Monteiro, J. Pereira, and L. Rodrigues. Integração do flight simulator 2002 com um protocolo de difusão epidémica. In *Actas da 6ª Conferência sobre Redes de Computadores*, Bragança, Portugal, September 2003.

[17] Alberto Montresor. A reliable registry for the jgroup distributed object model. In *Proceedings of the Third European Research Seminar on Advances in Distributed Systems (ERSADS '99)*, 1999.

[18] Alberto Montresor, Renzo Davoli, and Ozalp Babaoglu. Enhancing jini with group communication. *Distributed Computing Systems Workshops, International Conference on*, 0:0069, 2001.

[19] L. E. Moser, P. M. Melliar-smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39:54–63, 1996.

[20] N. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Transparent consistent replication of Java RMI objects. In *Proc. of the Int'l Symp. on Distributed Objects and Applications*. IEEE Computer Society, 2000.

[21] Object Management Group, Inc. Common object request broker architecture (v. 3.1). http://www.omg.org/technology/documents/vault.htm, 2008.

[22] Object Management Group, Inc. Fault tolerant CORBA (v. 1.0). http://www.omg.org/spec/FT, 2010.

[23] Sun Microsystems R. Srinivasan. Rpc: Remote procedure call protocol specification version 2. August 1995.

[24] Diogo Reis and Hugo Miranda. A framework for transparent fault-tolerant remote method invocation. In *Network Computing and Applications*, 2011.

[25] Diogo Reis and Hugo Miranda. Ftrmi: Fault-tolerant transparent rmi. In *Symposium on Applied Computing*, SAC'12. ACM press, 2012.

[26] Diogo Reis and Hugo Miranda. Ftrmi: Fault-tolerant transparent rmi. In *SAC '12 Proceedings of the 2012 ACM Symposium on Applied Computing*, page Aguardar publicação, 2012.

[27] Injong Rhee, Shun Yan Cheung, Phillip W. Hutto, Alan T. Krantz, and Vaidy S. Sunderam. Group communication support for distributed collaboration systems. *Cluster Computing*, 2:3–16, January 1999.

[28] Luís Rodrigues, Hugo Miranda, Ricardo Almeida, João Martins, and Pedro Vicente. The GlobData fault-tolerant replicated distributed object database. In Hassan Shafazand and A Min Tjoa, editors, *Proceedings of the EurAsia-ICT 2002: Information and Communication Technology*, number 2510 in Lecture Notes in Computer Science, pages 426–433, Shiraz, Iran, October 29–31 2002. Springer.

[29] Ann Wollrath Roger, Roger Riggs, and Jim Waldo. A distributed object model for the java system. *USENIX Computing Systems*, 9, 1996.

[30] Roger Sessions. *COM and DCOM: Microsoft's vision for distributed objects*. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[31] Sun Microsystems, Inc. Java remote method invocation specification (v. 5, r. 1.1). http://download.oracle.com/javase/6/docs/technotes/guides/rmi/index.html, 2004.

[32] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus: a flexible group communication system. *Commun. ACM*, 39:76–83, April 1996.

[33] Jim Waldo. Remote procedure calls and java remote method invocation. *IEEE Concurrency*, 6:5–7, July 1998.

[34] Pawel T. Wojciechowski, Sergio Mena, and André Schiper. Semantics of protocol modules composition and interaction. In F. Arbab and C. Talcott, editors, *Proceedings of The Fifth International Conference on Coordination Models and Languages*

(Coordination 2002), number 2315 in Lecture Notes in Computer Science, pages 389–404, York, UK, April 8–11 2002. Springer.

[35] A. F. Zorzo and R. J. Stroud. A distributed object-oriented framework for dependable multiparty interactions. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '99, pages 435–446, New York, NY, USA, 1999. ACM.