Combinando Objetos Distribuídos e Arquiteturas Orientadas a Eventos em uma Infra-estrutura de Comunicação para Sistemas Distribuídos

Aliandro Higino Guedes Lima

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Informática da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Informática.

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Redes de Computadores e Sistemas Distribuídos

Walfredo da Costa Cirne Filho

(Orientador)

Campina Grande, Paraíba, Brasil

Resumo

Nas últimas décadas, muito esforço foi empregado na intenção de tornar o desenvolvimento de sistemas distribuídos tão simples quanto o desenvolvimento de sistemas locais. Todavia, certas características inerentes a ambientes distribuídos (e.g., falhas parciais e concorrência), precisam ser evidenciadas para o programador. Existem também aspectos como conectividade parcial (devido ao uso de *firewalls* e NATs) que são bastante comuns quando a comunicação acontece entre múltiplos domínios administrativos. Diante disso, o modelo de Objetos Distribuídos surgiu como uma solução que se aproxima ao máximo do paradigma local orientado a objetos. Contudo, objetos distribuídos criam a ilusão de que as threads atravessam o espaço de endereçamento local, caminhando por toda a aplicação distribuída. Conseqüentemente, esta solução apresenta todos os problemas relacionados ao modelo complexo e não-determinístico de threads. Além disso, uma vez que objetos distribuídos usam um modelo de comunicação bloqueante, não são apropriados para aplicações nas quais um cliente tem outras "coisas" a fazer além de esperar uma resposta do servidor. Como alternativa, existe uma série de soluções baseadas em mensagens, que delimitam escopo para as threads da aplicação, mas falham em fornecer boa integração com a linguagem de programação, mecanismo de detecção de falhas bem definido e bom suporte à comunicação na presença de *firewalls* e NATs. Nesta dissertação, sustentamos a tese de que *é possível combinar objetos distribuídos com arquiteturas event-driven (tipo específico de solução baseada em mensagens), construindo uma solução bem integrada à linguagem de programação, apropriada para aplicações não-bloqueantes e cujas threads têm escopo bem definido*. Para sustentar nossa tese, apresentamos o JIC (*Java Internet Communication*), uma solução que também permite comunicação na presença de *firewalls* e NATs e oferece um mecanismo de detecção de falhas simples de usar e com semântica precisa. Experimentos mostram que o JIC tem desempenho comparável a Java RMI e uma análise que considera aspectos de engenharia de software mostra que, usando JIC, é possível focar mais em lógica de negócio e construir um código mais modularizado, explorando paralelismo sem precisar escrever código multi-threaded e, conseqüentemente, evitando problemas inerentes a threads.

Abstract

In the last decades, much work has been done in order to make the development of distributed systems as simple as the development of centralized systems. However, there are inherent characteristics of a distributed environment (e.g., partial failure and concurrency), which need to be explicited to the programmer. There are also other aspects, like partial connectivity (imposed by firewalls and NATs), which are common in communication across multiple administrative domains. Distributed Objects then appeared as a solution that is as close as possible to the centralized object oriented model. Nevertheless, distributed objects create the illusion that threads traverse the whole application, which brings all the problems related to the complex and non-deterministic thread model. Moreover, as distributed objects rely on a blocking communication model, they are not well suited for applications in which a client has other things to do besides waiting for the server response. As an alternative, there are a number of message-based solutions, which delimit scope for it's threads, but fail on providing good integration to the programming language, a well-defined failure detection mechanism or even a good support for firewall and NAT traversal. Our thesis is that *it is possible to combine distributed objects with an event-driven architecture (which is an specific type of message-based solution) in order to provide a solution that relies on a non-blocking communication model, yet providing close semantics to the object oriented paradigm, providing precise scope for the application's threads*. Then, we propose JIC (Java Internet Communication), a solution that supports our thesis, also designed to be firewall and NAT friendly and which provides an embedded failure detection mechanism simple to use and with precise semantics. An evaluation shows that JIC has performance comparable to Java RMI. A software engineering analysis also shows that, by using JIC, it is possible to focus more on the business logic of the application and build well-modularized code, exploiting parallelism without writing multi-threaded programs. This greatly avoids the inherent problems of threads.

Agradecimentos

Serão acrescentados à versão final.

Conteúdo

1 Introdução **1**

1.1 Motivação 2

1.2 JIC (*Java Internet Communication*) 5

1.3 Organização da Dissertação 7

2 Estado da Arte: Comunicação em Sistemas Distribuídos **8**

2.1 Comunicação Síncrona: Objetos Distribuídos 10

2.1.1 Java RMI 11

2.1.2 CORBA 12

2.2 Comunicação Assíncrona: Troca de Mensagens 13

2.2.1 *Sockets* 16

2.2.2 PVM e MPI 17

2.2.3 EBMs e MOMs 17

2.2.4 Twisted 19

2.3 Objetos Distribuídos Assíncronos 19

2.3.1 ARMI 19

2.3.2 Comunicação Assíncrona em CORBA 20

2.3.3 Comunicação Assíncrona no Twisted 21

2.4 Outras Preocupações Importantes em Sistemas Distribuídos 22

2.4.1 Detecção de Falhas 23

2.4.2 Comunicação na presença *Firewalls* e NATs 27

3 Definição do Problema **31**

3.1 O Problema com Threads 31

3.2 Threads e Objetos Distribuídos 33
3.3 Objetos Distribuídos e Aplicações não Bloqueantes 36

4 Solução proposta: *Java Internet Communication* **40**
4.1 Arquitetura 41
4.1.1 Access Point 41
4.1.2 Serviços e Objetos 43
4.1.3 Detecção de Falhas 44
4.2 Conectividade 46
4.3 Semântica 48
4.4 Modelo de Programação 52
4.4.1 Interfaces Remotas 53
4.4.2 Implementação de Interfaces Remotas 55
4.4.3 Exportando um Objeto JIC 56
4.4.4 Obtendo uma Referência Remota 57
4.4.5 Passagem de Parâmetros 59
4.4.6 Detecção de Falhas 60
4.5 Implementação 63
4.5.1 Linguagem de Programação 63
4.5.2 *Stubs* e Eventos 64
4.5.3 Semântica das Invocações 66
4.5.4 *CommunicationLayer* 67
4.5.5 Protocolo de Conexão 69
4.5.6 Detecção de Falhas 71

5 Avaliação da Solução **77**
5.1 Avaliação de Desempenho 78
5.1.1 Experimento 78
5.1.2 Métricas 80
5.1.3 Resultados 80
5.2 Avaliação de Engenharia de Software 85
5.2.1 Métricas 87

5.2.2 Resultados 91

6 Conclusões e Trabalhos Futuros **103**

6.1 Conclusões 103

6.2 Trabalhos Futuros 104

6.2.1 Evolução da implementação do JIC 104

6.2.2 Calibrar Mecanismo Atual de Detecção de Falhas 105

6.2.3 Estudar Efeitos do Mecanismo de Detecção Probabilístico 106

6.2.4 Comunicação Síncrona na Periferia da Aplicação 106

Lista de Figuras

2.1 Modelo *publish/subscribe*. 14
2.2 Modelo de monitoramento *push* 24
2.3 Modelo de monitoramento *pull* 25
2.4 Negociação de conexão padrão do TCP à esquerda. Na direita, temos a situação em que existe um *firewall* em ambas as máquinas. 28
2.5 Iniciação simultânea de conexão TCP à esquerda. Na direita, temos a situação em que existe um *firewall* em ambas as máquinas. 29

3.1 Código de duas aplicações `A` e `B`, respectivamente nas Figuras (a) e (b) . . . 33
3.2 Código de `A` modificado com bloco *synchronized* em suas chamadas 34
3.3 Código da aplicação `C`, cujo método `metodo_C` realiza controle de concorrência. 35

4.1 Arquitetura de um *Access Point* 42
4.2 Arquitetura Jabber. 47
4.3 Fluxo de comunicação entre o cliente Jabber C_1 e o cliente Jabber C_2 . . . 48
4.4 Cenário de Comunicação entre objetos A e B 50
4.5 Reuso da implementação remota oferecida pelo JIC 55
4.6 Implementação remota sem reuso da classe oferecida pelo JIC 56
4.7 Arquivo de configuração para uma aplicação JIC: *arquivo.conf*. 58
4.8 Campos de um evento JIC. 64
4.9 Campos de um `JICEventMetadata` 65
4.10 Padrão *cadeia de responsabilidades* implementado pela `CommunicationLayer`. 68
4.11 (a) Negociação da conexão iniciada pelo objeto A e (b) iniciação simultânea. 70

5.1 (a) Aplicação cliente-servidor para matriz pequena (ordem 10) e (b) para matriz grande (ordem 150) 81
5.2 (a) Aplicação *peer-to-peer* para matriz pequena (ordem 10) e (b) para matriz grande (ordem 150) 84
5.3 Padrão *event-driven* usado no OurGrid 2.2. 86

Lista de Tabelas

5.1 Características das versões do OurGrid analisadas. A última coluna indica se houve mudanças de funcionalidades em relação à versão anterior analisada. A versão 2.1.3 contém o conjunto inicial de funcionalidades. Detalhe: na tabela, RMI representa "Java RMI". Da mesma forma, "RMI + padrão" significa "Java RMI + padrão *event-driven*". 87

5.2 Comparação do tamanho do código entre as versões 2.1.3 e 2.2 do OurGrid. 94

5.3 Número de blocos synchronized nas versões 2.1.3 e 2.2 do OurGrid. 95

5.4 Espalhamento do código de detecção de falhas para a versão 2.1.3 do OurGrid. Cada entidade (MyGrid, Peer e GuM) representa um conjunto de objetos remotos em JVMs diferentes. 96

5.5 Espalhamento do código de detecção de falhas para a versão 2.2 do OurGrid. Cada entidade (MyGrid, Peer e GuM) representa um conjunto de objetos remotos em JVMs diferentes. 96

5.6 Porção da aplicação que lida com preocupações transversais às funcionalidades do OurGrid 2.2. Cada linha representa um componente do padrão *event-driven* 98

5.7 Complexidade das classes na versão 2.1.3 e 2.2 98

5.8 Comparação do tamanho do código entre as versões 3.3 e 4.0 do OurGrid. . 100

5.9 Número de blocos synchronized nas versões 3.3 e 4.0 do OurGrid. 100

5.10 Espalhamento do código de detecção de falhas para a versão 3.3 do OurGrid. Cada entidade representa um conjunto de objetos remotos em JVMs diferentes. 101

5.11 Complexidade das classes na versão 2.1.3 e 2.2 102

Capítulo 1

Introdução

Nas últimas décadas, muito esforço foi empregado na intenção de tornar o desenvolvimento de sistemas distribuídos tão simples quanto o desenvolvimento de sistemas locais. Basicamente, *sistemas distribuídos* são formados por componentes que executam em processos distintos, potencialmente em máquinas distintas, cuja comunicação se dá através da troca de mensagens. Em contrapartida, *sistemas locais* são caracterizados por possuírem componentes que executam em um mesmo processo, ou seja, em um mesmo espaço de endereçamento, de forma que a comunicação se dá através de acesso direto aos componentes. Componentes que executam em espaços de endereçamento distintos são ditos "componentes remotos". De maneira análoga, componentes que dividem um mesmo espaço de endereçamento são chamados "componentes locais".

Entretanto, programar sistemas distribuídos não pode ser tão simples quanto programar sistemas locais [WWWK94]. Isto se deve às diferenças relacionadas aos ambientes em que tais sistemas executam. De fato, existem características fundamentais de ambientes distribuídos que não estão presentes em ambientes locais. Dentre tais características, ressaltamos a necessidade de lidar com concorrência e a presença de falhas parciais. Além disso, a existência de conectividade parcial devido ao uso de *firewalls* e NATs [1] é bastante comum quando a comunicação acontece entre diferentes domínios administrativos. Isso significa que construir um modelo de programação distribuído que não leva em consideração as características destes ambientes é estar propenso ao fracasso. Logo, a tentativa de unificar os modelos de programação local e distribuído, no melhor caso, faria o modelo de programação local tão

[1] Direcionamos o leitor à Seção 2.4.2, para uma explicação sobre o funcionamento de *firewalls e NATs*

complexo quanto o distribuído [WWWK94].

1.1 Motivação

Diante desta necessidade de se considerar as diferenças entre sistemas distribuídos e locais, um grande progresso foi conseguido através do modelo de Objetos Distribuídos [CD05], discutido em detalhes no Capítulo 2. Tal modelo é uma extensão do paradigma de programação orientado a objetos para um ambiente distribuído, de forma que objetos localizados em diferentes processos podem se comunicar através de invocações remotas de métodos. Muitas soluções, como Java RMI [Gro02; RMI06] e CORBA [OMG04], fornecem *middlewares* que escondem os detalhes de baixo nível relacionados a processos e trocas de mensagens, oferecendo ao programador um modelo de programação o mais próximo possível do paradigma orientado a objetos. Além de fornecer um modelo próximo à programação local, o modelo de objetos distribuídos permite uma boa integração com a linguagem de programação, uma vez que o mecanismo de checagem de tipos da linguagem também funciona entre objetos remotos. O uso de tal mecanismo entre objetos remotos ajuda o programador a capturar *bugs* em estágios iniciais do desenvolvimento de uma aplicação.

A proximidade de objetos distribuídos em relação ao paradigma orientado a objetos é conseguida através de um modelo de comunicação *bloqueante*. Neste modelo, um *cliente* invoca um *servidor* e bloqueia sua thread até receber uma resposta ou detectar a falha do servidor. Este modelo permite uma abordagem simples em relação à detecção de falhas parciais, uma vez que o *middleware* pode usar os *timeouts* da camada de comunicação para detectar uma possível falha e desbloquear a thread do cliente através do lançamento de uma exceção. Dessa forma, a detecção de falhas é acoplada à invocação e tratada de maneira análoga aos erros que acontecem em sistemas locais.

No modelo de objetos distribuídos, uma vez que a thread do cliente bloqueia, tem-se a ilusão de que esta thread atravessa o espaço de endereçamento local e executa por outras partes do sistema distribuído, até retornar um resultado ou uma notificação de falha. Na realidade, cada invocação remota é executada no servidor por uma nova thread. Uma vez que o servidor usa uma nova thread para cada invocação recebida e sistemas distribuídos são potencialmente acessados simultaneamente por vários componentes, o código do lado

servidor precisa ser multi-threaded. Isso significa que o programador deve lidar com código concorrente ao construir um objeto que será acessado de maneira remota por mais de um componente.

Todavia, conforme descrito no Capítulo 3, o modelo de threads tem se mostrado problemático para programação concorrente. Em primeiro lugar, raciocinar sobre programas multi-threaded é muito difícil, visto que o modelo de threads é não-determinístico [Her05; Edw06]. Além disso, existe um problema relacionado à componentização em programas multi-threaded que usam mecanismos para realizar exclusão mútua (*locks* [And01]). Não se pode combinar dois componentes que usam *locks* corretamente e saber se o resultado desta combinação está correto sem analisar ambos os códigos [Her05]. Isso vai contra uma das técnicas mais básicas de engenharia de software, que consiste em modularizar funcionalidades em componentes reusáveis, de forma que um componente possa usar outros desde que conheça suas interfaces, sem precisar conhecer suas implementações.

Se o modelo de threads se mostra complicado para aplicações concorrentes locais, a situação se torna ainda mais complexa quando consideramos aplicações concorrentes distribuídas. A Seção 3.2 mostra que desenvolver código baseado em objetos distribuídos é muito complexo pelos seguintes motivos: (i) raciocinar sobre threads em um ambiente distribuído é ainda mais difícil do que em ambientes locais; (ii) é preciso raciocinar sobre threads para toda a aplicação, uma vez que objetos distribuídos fornecem a ilusão de continuação da thread do cliente; (iii) o problema de componentização de *locks* é mais grave em um ambiente distribuído do que em um ambiente local e (iv) o suporte para depuração de aplicações multi-threaded em sistemas locais não funciona da mesma forma para sistemas distribuídos.

Além dos problemas impostos pelo modelo de threads, o uso de objetos distribuídos possui uma outra limitação. Uma vez que objetos distribuídos usam um modelo de comunicação bloqueante, um cliente não pode prosseguir com processamento local enquanto uma requisição remota está sendo executada (a não ser que o cliente seja multi-threaded). Porém, existem aplicações em que o cliente tem outras coisas a fazer além de bloquear à espera de uma resposta do servidor (e.g., aplicações interativas e aplicações paralelas). Para estas aplicações, portanto, o paradigma de objetos distribuído não é apropriado. Conforme veremos na Seção 3.3, maneiras óbvias de adaptar objetos distribuídos a este tipo de aplicação, fornecendo a ilusão de invocações não-bloqueantes, apresentam sérias desvantagens.

Uma maneira possível de lidar com os problemas inerentes a objetos distribuídos consiste em buscar alternativas a este paradigma, evitando seu uso. Soluções baseadas em mensagens são a alternativa natural para objetos distribuídos. Existem diversas possibilidades, desde o uso de *sockets*, até soluções mais sofisticadas, como *middlewares* baseados em eventos (*Event-Based Middleware* - EBM) [Pie02; SCT95] e *middlewares* orientados a mensagens (*Message-Oriented Middleware* - MOM) [IEE06; MHC00], passando por abordagens para programação paralela, como as bibliotecas PVM (*Parallel Virtual Machine*) [Sun90] e MPI (*Message Passing Interface*) [For94]. Tais soluções não bloqueiam à espera de um resultado. Entretanto, não são tão bem integradas à linguagem de programação como as soluções baseadas em objetos distribuídos.

Em 2005, precisávamos de uma solução para comunicação não-bloqueante bem integrada à linguagem de programação, que não apresentasse os problemas relacionados a threads e ainda assim fosse leve e apropriada para ambientes compostos por múltiplos domínios administrativos (onde há tipicamente a presença de *firewalls* e NATs) e nos quais situações de falhas são comuns. Tal necessidade surgiu dentro do projeto OurGrid, um *grid* computacional *peer-to-peer* de livre acesso, no qual laboratórios de pesquisa podem doar seus recursos computacionais ociosos uns aos outros [CBA+06]. O software OurGrid era baseado em Java RMI e, apesar de estar em produção desde Dezembro de 2004, os problemas decorrentes do uso de objetos distribuídos tornaram o código bastante frágil, dificultando em demasia sua manutenção e evolução [DCS06].

Diante disso, estava claro que era necessária uma solução baseada em mensagens, evitando assim o uso de objetos distribuídos. Entretanto, não foi possível encontrar nenhuma solução baseada em mensagens que se adequasse às necessidades do OurGrid. De fato, as soluções existentes foram criadas com outros objetivos em mente. *Sockets* objetivam expor a camada de transporte ao programador da aplicação. EBMs e MOMs desacoplam os componentes de um sistema distribuído através do paradigma *publish/subscribe*, geralmente incrementado por serviços como *store-and-forward*, roteamento de mensagens e gerenciamento de transações. PVM e MPI, por sua vez, foram criados para facilitar operações coletivas, simplificando o desenvolvimento de aplicações paralelas. Apesar de evitar grande parte dos problemas com threads, nenhuma das soluções oferece um bom suporte para atravessar *firewalls* e NATs ou oferece mecanismos flexíveis e sofisticados para detecção de falhas.

Além disso, nenhuma delas é tão bem integrada à linguagem de programação quanto objetos distribuídos.

Um paradigma de programação que pode ser usado para delimitar o escopo das threads de uma aplicação é baseado na arquitetura *event-driven* [WCB01]. Trata-se de um tipo particular de solução baseada em mensagens, de forma que as mensagens recebidas são armazenadas em uma fila e consumidas por threads próprias da aplicação (*event-loops*) que trabalham de forma iterativa, retirando uma mensagem da fila e disparando uma chamada a um *event-handler*, que processa a mensagem. No contexto de arquiteturas *event-driven*, cada mensagem representa um evento que deve ser tratado pela aplicação.

Através deste paradigma, é possível modularizar uma aplicação, de forma que cada componente do sistema é um objeto ativo, que possui sua própria thread de execução, a qual executa as requisições remotas que foram recebidas [SRSS00]. Sendo assim, pode-se programar uma aplicação em que cada módulo tem seu escopo bem definido e as threads de um módulo não interferem em um outro módulo. Diante desta arquitetura, o desenvolvedor que constrói uma aplicação sobre uma arquitetura *event-driven* é levado a pensar na aplicação de maneira diferente do modelo tradicional de programação seqüencial. Um programa passa a ser estimulado por eventos, de forma que o programador precisa pensar em *que eventos devem ser gerados*, *que eventos devem ser manipulados pela aplicação* e *de que maneira a aplicação responde quando um evento acontece*.

Soluções baseadas em mensagens e construídas através de uma arquitetura *event-driven*, delimitando o escopo de suas threads, simplesmente não possuem os problemas relacionados ao modelo de threads discutidos no Capítulo 3. Contudo, tais soluções não são tão amplamente usadas quanto as soluções construídas sobre objetos distribuídos. Atribuímos o uso reduzido da solução *event-driven* ao fato de que, assim como as demais soluções baseadas em mensagens, esta solução não é tão bem integrada à linguagem como objetos distribuídos.

1.2 JIC (*Java Internet Communication*)

Ao longo deste capítulo, vimos que (i) o paradigma de objetos distribuídos possui sérios problemas devido ao modelo de threads; além disso, (ii) objetos distribuídos não são apropriados para aplicações não-bloqueantes, uma vez que a thread que realiza a invocação bloqueia à

espera de um resultado ou de uma exceção que indique a falha do componente remoto; vimos também que (iii) a arquitetura *event-driven* elimina os problemas relacionados ao modelo de threads, mas não é tão bem integrada à linguagem de programação.

Acreditamos que é possível construir um modelo de programação bem integrado à linguagem de programação sobre a arquitetura *event-driven*, aproveitando uma solução existente. Portanto, nesta dissertação, desafiamos o argumento de que soluções baseadas em mensagens, mais precisamente soluções *event-driven*, precisam oferecer primitivas de baixo nível (*send/receive*) ao programador, não sendo bem integradas à linguagem de programação. Consideramos que ser bem integrado à linguagem consiste em permitir que o mecanismo de checagem de tipos funcione para interações entre componentes distribuídos. Logo, podemos formular a seguinte tese, a qual desejamos confirmar através desta dissertação.

Tese: *é possível combinar objetos distribuídos com arquiteturas event-driven, construindo uma solução bem integrada à linguagem de programação, apropriada para aplicações não-bloqueantes e cujas threads têm escopo bem definido, minimizando os principais problemas de ambas as abordagens*.

Com o objetivo de confirmar esta tese, bem como de satisfazer as necessidades do OurGrid, desenvolvemos o JIC (*Java Internet Communication*), uma infra-estrutura de comunicação para sistemas distribuídos que (i) é *event-driven*; (ii) usa um modelo de comunicação não-bloqueante, ainda assim provendo semântica próxima do paradigma orientado a objetos; (iii) cria escopo bem definido para suas threads; (iv) lida bem com ambientes em que há presença de *firewalls* e NATs e (v) fornece um mecanismo de detecção de falhas simples de usar e com semântica precisa.

Nossos resultados mostram que o JIC possui desempenho comparável a Java RMI, exemplo representativo e bem sucedido de *middleware* baseado em objetos distribuídos. Por outro lado, em termos de engenharia de software, o uso do JIC resulta em várias implicações positivas no desenvolvimento de aplicações distribuídas. Dentre os benefícios observados, podemos citar que o programador se desvia menos da lógica de negócio da aplicação. Além disso, usando JIC, o programador não precisa lidar com os problemas inerentes ao modelo de threads, pois é possível construir uma aplicação distribuída completa sem código multithread. Por fim, caso seja preciso raciocinar sobre threads, o programador pode limitar

o escopo sobre o qual raciocina, uma vez que as threads da aplicação possuem fronteiras bem definidas. A análise realizada mostra que a implantação do JIC no OurGrid diminuiu o tamanho do código em mais de 40%, eliminando código que não fazia parte da lógica de negócio e antes representava em torno 20% da aplicação. Além disso, o número de blocos `synchronized`, que tratam situações onde há concorrência, diminuiu em mais de 60%, minimizando a preocupação do programador com código multi-threaded.

1.3 Organização da Dissertação

Este capítulo apresentou o problema tratado nesta dissertação, porém, sem uma grande riqueza de detalhes, pois o principal intuito é o de motivar a dissertação e definir quais os seus objetivos. O restante desta dissertação está organizado da seguinte maneira:

No Capítulo 2, apresentamos o estado da arte em comunicação para sistemas distribuídos, contextualizando o leitor sobre as principais soluções que existem para os aspectos relevantes em sistemas distribuídos.

Feito isso, o leitor já estará familiarizado com as abordagens existentes para cada aspecto. Este conhecimento permitirá que no Capítulo 3 o problema atacado nesta dissertação seja definido com mais detalhes, e que seja explicado o porquê de cada solução apresentada no Capítulo 2 não resolver nosso problema.

Adiante, no Capítulo 4, apresentamos o JIC. Começamos com detalhes arquiteturais, de alto nível, associando as decisões de projeto mais importantes à maneira como tais decisões nos ajudam a resolver aspectos do problema definido. Em seguida, adentramos no modelo de programação, mais rico em detalhes e finalmente nos aprofundamos nos detalhes mais importantes relacionados à implementação.

No Capítulo 5, fazemos uma avaliação experimental do JIC em contraste com Java RMI. Esta avaliação consiste em uma análise de desempenho e em uma análise de código considerando algumas métricas relacionadas à engenharia de software.

Por fim, no Capítulo 6, concluímos a dissertação e apresentamos sugestões para trabalhos futuros.

Capítulo 2

Estado da Arte: Comunicação em Sistemas Distribuídos

Em sistemas distribuídos, uma vez que há comunicação entre objetos remotos, existe um aspecto extra que deve ser considerado ao permitir tal comunicação. Trata-se do conceito de *localização*. Objetos remotos diferem de objetos locais por estarem localizados em espaços de endereçamento diferentes. Porém, o paradigma orientado a objetos (ou modelo de programação orientado a objetos) não considera localização como parte da especificação de um objeto. Localização costuma ser vista como um detalhe de implementação. De fato, visto que sistemas locais oferecem acesso direto ao endereço de memória no qual um objeto está localizado, a implementação de objetos locais torna o conceito de localização bastante transparente para o programador.

Logo, segundo o modelo de programação orientado a objetos (que não considera localização), objetos locais e remotos possuem as mesmas características. Sendo assim, tal paradigma se adequa, em teoria, tanto a sistemas locais como a sistemas distribuídos. Mas na prática, modelos de programação locais não se adequam a sistemas distribuídos [WWWK94], pois estes últimos são mais complexos devido às particularidades observadas em seu ambiente de execução (e.g., concorrência e falhas parciais). Como conseqüência, os modelos locais de programação foram estendidos para ambientes distribuídos, gerando novos modelos [CD05], como por exemplo:

1. **Chamada de Procedimento Remoto:** conhecido pela abreviatura RPC (*Remote Pro-*

cedure Call), representa uma especialização do modelo de programação procedural para um ambiente distribuído. Permite que programas clientes realizem chamadas de procedimentos em programas servidores, os quais executam em processos separados, muito possivelmente em máquinas distintas;

2. **Invocação Remota de Método:** conhecido pela abreviatura RMI (*Remote Method Invocation*), representa uma especialização do modelo de programação orientada a objetos para um ambiente distribuído. Este modelo permite que objetos realizem chamadas de métodos em outros objetos que residem em outros processos, muito possivelmente em outras máquinas.

Além das especializações dos modelos locais, é possível realizar comunicação simplesmente através da *troca de mensagens*. Basicamente, este modelo consiste em usar primitivas de envio e recebimento (*send/receive*), no qual uma entidade pode enviar mensagens para uma ou mais entidades, desde que estejam usando um mesmo protocolo (e.g., TCP, através de *sockets*).

Dentre os modelos para sistemas distribuídos exemplificados acima, o modelo de *invocação remota de métodos* e o modelo de *troca de mensagens* são amplamente usados para realizar comunicação *síncrona* e *assíncrona*, respectivamente. Os conceitos de *síncrono* e *assíncrono* podem ter várias interpretações diferentes em sistemas distribuídos. Por isso, considerando o escopo desta dissertação, usaremos estes termos com os significados apresentados a seguir, exceto quando explicitamente indicado de outra forma.

Em uma interação **síncrona** entre duas entidades remotas (e.g., chamada remota de método), a thread que executa a invocação (thread do cliente) bloqueia até o término da execução remota. Conseqüentemente, o cliente não pode progredir com computação local enquanto a chamada remota está sendo executada, a menos que seja um cliente multi-threaded.

Por outro lado, em interações **assíncronas**, a thread do cliente não bloqueia à espera do término da operação remota, de forma que é possível realizar processamento local em paralelo com as operações remotas em execução.

2.1 Comunicação Síncrona: Objetos Distribuídos

O modelo de invocação remota de métodos (RMI), também conhecido como *modelo de objetos distribuídos*, permite que invocações remotas sejam realizadas de maneira muito parecida com invocações locais. Para que isto seja possível, é preciso que o mecanismo de checagem de tipos da linguagem funcione entre duas entidades remotas. Esta boa integração com a linguagem é conseguida através de entidades intermediárias, chamadas *stubs*. *Stubs* são objetos locais que representam um objeto remoto. Ao mesmo tempo em que possuem a mesma interface do objeto que representam, conhecem detalhes sobre comunicação que os permitem encaminhar invocações recebidas para o objeto alvo. Note que tal integração com a linguagem em sistemas distribuídos evita que o programador precise lidar com primitivas de comunicação de mais baixo nível (e.g., primitivas *send*/*receive* para troca de mensagens entre duas entidades remotas).

Basicamente, programar um objeto remoto consiste em: (i) definir interfaces remotas, as quais serão implementadas pelo objeto remoto, (ii) construir a implementação das interfaces remotas definidas e (iii) tornar acessível o objeto que implementa as interfaces remotas. Um cliente deste objeto deve: (i) recuperar uma referência para o objeto remoto (*stub*) e (ii) invocar métodos nesta referência.

No modelo de objetos locais, a próxima linha de código só é executada quando a linha de código atual tiver sua execução concluída. No modelo de objetos distribuídos, se a linha de código atual representa uma invocação remota, a próxima linha de código só poderá ser executada quando a chamada remota for concluída. Portanto, uma vez que o modelo de objetos distribuídos busca manter-se o mais próximo possível do modelo de objetos locais, a comunicação acontece naturalmente de maneira síncrona. Dessa forma, a thread do objeto que realiza uma invocação remota bloqueia até que o resultado seja recebido (ou uma falha seja detectada). Enquanto a thread local está bloqueada, uma nova thread é criada no lado remoto para executar a invocação.

Conforme mencionado no Capítulo 1, este processo fornece ao programador a ilusão de que a thread do objeto cliente atravessa seu espaço de endereçamento e continua sua execução no lado servidor. Realmente, a nova thread criada no servidor representa a continuação da thread do cliente. Por causa desta característica, o servidor precisa ser multi-threaded e o

escopo das threads de uma aplicação passa a ser toda a aplicação distribuída, uma vez que as threads locais têm a "liberdade" de continuar em outros espaços de endereçamento.

A abordagem síncrona, porém, facilita o tratamento de falhas das entidades remotas. Visto que a thread do objeto cliente está bloqueada, falhas detectadas pela infra-estrutura de comunicação podem ser repassadas ao cliente através do lançamento de uma exceção, exatamente como aconteceria no modelo local, caso um problema acontecesse na execução de um método. Em contrapartida, soluções para objetos distribuídos costumam usar os próprios *timeouts* da camada de comunicação para detectar falhas, de forma que não usam mecanismos sofisticados para detecção de falhas, como os descritos na Seção 2.4.1. No que diz respeito à comunicação na presença de *firewalls* e NATs, soluções baseadas em objetos distribuídos costumam exigir que haja uma porta aberta (ou tradução reversa no NAT) para cada objeto remoto da aplicação. Esta abordagem vai de encontro aos princípios administrativos da maioria dos domínios, cuja idéia consiste em evitar abrir portas nos *firewalls*. Diante desta situação, alguns *middlewares* (e.g., Java RMI) oferecem alternativas, como o tunelamento HTTP [And02], que exige apenas a porta HTTP aberta (situação bastante comum). Entretanto, através desta alternativa, é preciso usar um componente extra, que recebe as requisições HTTP e as converte em requisições de aplicação, o que pode degradar o desempenho da mesma.

As principais e mais usadas soluções baseadas em objetos distribuídos são Java RMI [Gro02; RMI06] e CORBA [OMG04]. Ambas compartilham as características acima citadas, mas possuem algumas diferenças, que serão apresentadas a seguir.

2.1.1 Java RMI

Como o próprio nome indica, Java RMI fornece uma implementação em Java para objetos distribuídos. Por ser uma solução dependente de linguagem, não permite que objetos desenvolvidos em linguagens diferentes se comuniquem, ou seja, não possui interoperabilidade em relação à linguagem de programação. Por outro lado, em Java RMI, a semântica do modelo de objetos distribuídos foi construída sobre a semântica do modelo de objetos Java. Uma vez que a semântica foi baseada em uma linguagem específica, é possível integrar-se o máximo possível a esta linguagem, pois o sistema inteiro executa em um ambiente homogêneo (a máquina virtual Java). Por exemplo, a especificação do modelo de objetos remotos Java não precisa definir modos de passagem de parâmetros (parâmetros `in`, `out` ou `inout`),

pois o modelo local de Java considera um único modo padrão (todos os parâmetros são de entrada, i.e., parâmetros `in`). Além disso, o modelo remoto de objetos Java pode definir um mecanismo de *garbage collection* remoto, uma vez que *garbage collection* faz parte da especificação de objetos Java. Se o objetivo de Java RMI fosse interoperar entre linguagens de programação, não seria possível incluir *garbage collection* no modelo, pois há linguagens que não implementam este mecanismo (e.g., C++).

A definição de interfaces remotas em Java RMI é feita através da própria linguagem Java. Em [Mah02], é argumentado que o fato de não ser preciso aprender uma nova linguagem para definir as interfaces remotas dos objetos distribuídos contribui para que o desenvolvimento de aplicações distribuídas seja mais rápido através de Java RMI, quando comparado a outras soluções, tais como CORBA.

2.1.2 CORBA

CORBA (*Common Object Request Broker Architecture*) é uma especificação, não uma implementação. Trata-se de um padrão desenvolvido pelo OMG (*Object Management Group*) [OMG06a] que define o comportamento de objetos distribuídos. As implementações da especificação são feitas por ORBs (*Object Request Brokers*), os quais fornecem *middlewares* que dão suporte a aplicações baseadas em objetos distribuídos. CORBA foi desenvolvido para ser independente de linguagem de programação, permitindo que objetos distribuídos implementados em linguagens diferentes possam se comunicar de maneira transparente. Diante disso, as especificações de CORBA são neutras em relação à linguagem de programação.

Uma vez que CORBA visa interoperabilidade, a definição de interfaces remotas também deve ser feita de maneira neutra. Para isso, CORBA define uma linguagem própria, chamada IDL (*Interface Definition Language*, ou seja, Linguagem de Definição de Interface). Apesar de o programador precisar aprender uma nova linguagem, existe um argumento de que IDL é fácil de aprender, uma vez que representa uma linguagem descritiva [Cur97].

Os fornecedores de ORBs para uma determinada linguagem devem oferecer mapeamentos que permitam adequá-la à linguagem padrão IDL. Por isso, existem especificações feitas pelo OMG que permitem realizar o mapeamento de IDL para uma linguagem de programação e vice-versa. Considere um *middleware* que implemente o padrão CORBA e permita a construção de aplicações distribuídas através de Java. Ao compararmos interfaces Java a

interfaces IDL, veremos que existem incompatibilidades entre o modelo de objetos Java e o modelo descrito pela IDL. Por exemplo, IDL define parâmetros de saída (`out` e `inout`), os quais não existem em Java. O ORB, por sua vez, precisa lidar com esta incompatibilidade para permitir que objetos Java interoperem com outras linguagens. Em geral, os ORBs usam objetos internos chamados `Holders` que realizam este mapeamento automaticamente, de maneira transparente ao programador. Em outras situações, a incompatibilidade entre modelos é mais grave. A especificação do mapeamento entre IDL e Java [OMG06b] não define tipos correspondentes em Java para o tipo `long double`, presente em IDL. Da mesma forma, existem tipos sem sinais em IDL, que também não possuem correspondentes em Java, mas são associados aos tipos com sinais através da especificação IDL/Java. Por isso, o ORB que implementa uma linguagem específica deve lidar com todos os detalhes que diferenciam a linguagem e o modelo de CORBA, tentando aproximar os dois modelos ao máximo.

2.2 Comunicação Assíncrona: Troca de Mensagens

Existem várias soluções que usam o modelo de troca de mensagens (através de primitivas *send/receive*) para realizar comunicação. As alternativas variam desde *sockets*, passando por PVM [Sun90] e MPI [For94] até soluções como *Middlewares Orientados a Mensagens* (MOMs) [IEE06; MHC00] e *Middlewares Baseados em Eventos* (EBM) [Pie02; SCT95], além de soluções como o Twisted [Fet06], que fornece tanto um modelo de troca de mensagens, como de objetos distribuídos assíncronos. Todas estas soluções possibilitam comunicação assíncrona. Entretanto, diferem umas das outras em vários aspectos, apresentados a seguir:

1. **Endereçamento:** classificamos o estilo de endereçamento em duas categorias: (i) endereçamento *direto*, em que uma entidade conhece o endereço da entidade destino e envia mensagens diretamente para tal entidade, e (ii) endereçamento através do modelo *publish/subscribe*. Neste modelo, mostrado na Figura 2.1, existem três tipos de entidades: *serviço de eventos*, *objetos de interesse* e *objetos receptores*. O serviço de eventos é a entidade responsável pelo gerenciamento de eventos, notificando as entidades interessadas sobre o acontecimento dos mesmos. É também responsável por cadastrar e remover tipos de eventos, e gerenciar as entidades interessadas. Um objeto de in-

teresse registra um determinado tipo de evento no serviço de eventos, notificando-o sobre cada ocorrência do mesmo. Objetos receptores cadastram-se como interessados na ocorrência de um determinado tipo de evento [BGT+01] e são notificados sobre o acontecimento dos mesmos pelo serviço de eventos;

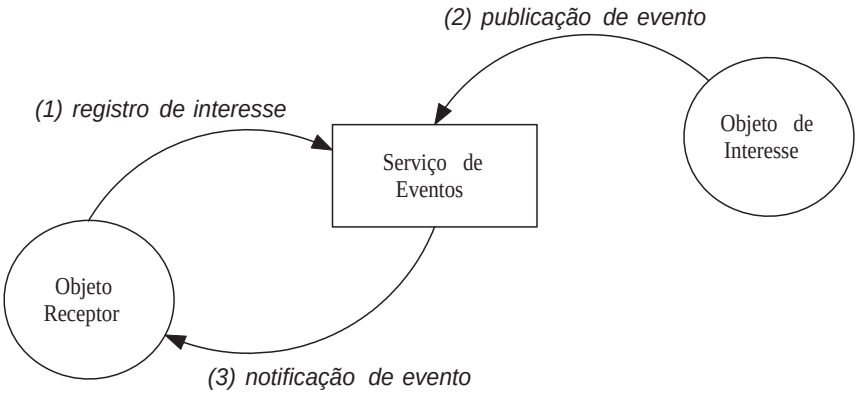


Figura 2.1: Modelo *publish/subscribe*.

2. **Dependência de tempo:** este aspecto considera a necessidade de as duas entidades que se comunicam precisarem estar rodando (*online*) ao mesmo tempo para que a comunicação aconteça. Se for necessário, dizemos que **existe dependência de tempo** entre as entidades. Caso contrário, dizemos que **não existe dependência de tempo** entre as mesmas. Em geral, este segundo caso é implementado através de um mecanismo conhecido como *store-and-forward*, no qual as mensagens endereçadas para uma entidade que está *offline* ficam armazenadas em um componente intermediário que participa da comunicação (e.g., um servidor). Quando a entidade está *online* novamente, este componente de comunicação encaminha as mensagens armazenadas que dizem respeito a esta entidade para a mesma, ou a entidade requisita o envio de tais mensagens;

3. **Integração com a linguagem:** assim como mencionado na Seção 2.1, dizemos que uma solução **é bem integrada à linguagem de programação** quando permite que o mecanismo de checagem de tipos da linguagem funcione entre entidades remotas. Soluções que oferecem ao programador apenas a possibilidade de executar primitivas de baixo nível para comunicação entre processos (e.g., *send* e *receive*) **não são tão bem integradas à linguagem de programação** quanto as soluções que permitem o

funcionamento do mecanismo de checagem de tipos;

4. **Comunicação em grupo**: Quando a comunicação entre entidades assume um modelo em que só é possível realizar o envio de mensagens processo-a-processo (*unicast*) [And02], dizemos que a solução **não implementa comunicação em grupo**. Caso contrário, se uma mensagem pode ser endereçada simultaneamente a várias entidades, i.e., uma mensagem endereçada a um grupo de processos (*multicast*) [And02], dizemos que a solução **implementa comunicação em grupo**;

5. **Programação *event-driven*:** O modelo de programação *event-driven*, assim como caracterizado na Seção 1.1, é composto por um *event-loop*, no qual uma ou mais threads permanecem em *loop* infinito, consumindo eventos de uma fila e bloqueando na mesma à espera de mais eventos. Caso uma solução processe suas mensagens através de um *event-loop*, dizemos que **implementa o modelo de programação *event-driven***. Se uma solução obrigatoriamente processa suas mensagens através de um *event-loop*, dizemos que tal solução **força programação event-driven**. Soluções que não implementam este modelo de programação ainda assim podem optar por construí-lo sobre o modelo de troca de mensagens.

6. **Detecção de falhas:** Aplicações distribuídas precisam estar preparadas para lidar com falhas parciais de componentes remotos. Para isso, é necessário um mecanismo de detecção de falhas. Dentre as abordagens possíveis, uma aplicação pode **não oferecer quaisquer mecanismos**, deixando o tratamento de falhas para a aplicação, **oferecer um mecanismo baseado em *timeouts* da camada de comunicação**, que são dependentes da infra-estrutura de comunicação e pouco configuráveis, **oferecer um mecanismo flexível e configurável**, no qual pode-se configurar parâmetros de qualidade de serviço da detecção e estender o mecanismo para usar estratégias novas de monitoramento, e por último, uma solução pode simplesmente **considerar que entidades nunca falham**, armazenando mensagens para serem entregues a uma entidade que saiu do ar quando a mesma reaparecer. Note que esta última abordagem apenas é possível caso não exista dependência de tempo, através de um mecanismo de *store-and-forward*, apresentado no item 2;

7. **Uso de entidades intermediárias:** Algumas soluções, para facilitar a comunicação na presença de *firewalls* e NATs, usam entidades intermediárias para enviar mensagens. Tais entidades são conhecidas como *relays* e o uso destas entidades evita que mais de uma porta precise ser aberta no *firewall* de um domínio administrativo. Logo, uma solução pode **usar *relays*** ou **não usar *relays*** em sua comunicação.

Considerando os aspectos descritos acima, comparamos a seguir as soluções para comunicação por troca de mensagens apresentadas no início desta seção:

2.2.1 *Sockets*

Sockets simplesmente expõem a camada de transporte para o programador da aplicação, que lida diretamente com primitivas *send* e *receive*. Por isso, não são bem integrados à linguagem. As mensagens enviadas usam endereçamento direto, sendo possível tanto a comunicação processo-a-processo, como comunicação em grupo. Visto que a comunicação não possui componentes intermediários, existe dependência de tempo entre as entidades. Por fim, este modelo não força que a comunicação aconteça seguindo um modelo *event-driven*, apesar de ser possível criar threads que recebem mensagens, as processam e bloqueiam à espera de uma nova mensagem.

Programar diretamente sobre *sockets* significa desenvolver um servidor que "escuta" em uma porta TCP (através da primitiva *receive*) e um cliente que envia dados (através da primitiva *send*). Porém, *sockets* TCP realizam comunicação orientada a conexão, de forma que duas entidades só podem se comunicar depois de estabelecer tal conexão. Um grande problema relacionado à comunicação sobre TCP (seja diretamente sobre *sockets* ou sobre qualquer modelo de programação que usa TCP como infra-estrutura) está no fato de que portas fechadas em *firewalls* não permitem que a conexão entre duas entidades separadas pelo *firewall* possa ser iniciada por qualquer um dos lados. Em geral, *firewalls* permitem que a conexão seja iniciada apenas pela entidade que está dentro do domínio protegido por eles. Sendo assim, ao programar em nível de *sockets*, caso uma entidade precise estabelecer conexão com uma entidade remota, esta última precisa estar associada a uma porta aberta no *firewall*, ou que possua tradução reversa no NAT em caso de redes privadas, como veremos na Seção 2.4.2. Em relação à detecção de falhas, a comunicação via *sockets* depende

dos *timeouts* definidos pela especificação TCP, ou configurados pela implementação em uso. Após estabelecida uma conexão TCP entre duas entidades, caso a conexão permaneça ociosa por um tempo pré-definido (segundo o RFC 1122 [IET06b], o valor padrão é duas horas), um pacote *keepalive_probe* é enviado para o lado remoto. Se o lado remoto não responder a este pacote, a falha da entidade remota é detectada.

2.2.2 PVM e MPI

PVM (*Parallel Virtual Machine*) e MPI (*Message Passing Interface*) foram desenvolvidas para possibilitar operações coletivas, facilitando o desenvolvimento de aplicações paralelas. Permitem endereçamento direto, sendo possível realizar tanto *multicast* como *unicast*. As primitivas de troca de mensagens são baseadas em *send* e *receive*, de forma que a integração com a linguagem não é tão boa quanto em objetos distribuídos. Assim como *sockets*, existe dependência de tempo entre as entidades e o modelo de comunicação não força o uso do modelo *event-driven*.

Tolerância a falhas em aplicações baseadas em MPI 1.x usam um modelo estático. A abordagem consiste em, se um elemento do grupo falha, o grupo inteiro é invalidado e a aplicação é finalizada. Já PVM e MPI 2.0 possuem um mecanismo de notificação que permite a construção de aplicações tolerantes a falhas. Através do mecanismo de PVM, por exemplo, cada processo possui um *deamon* (*pvmd*), que é capaz de detectar a falha de um outro processo se a comunicação com outro *pvmd* mantiver-se ociosa durante um *timeout* pré-definido. O problema de *firewalls* e NATs possui menos importância para aplicações construídas sobre MPI e PVM, uma vez que são soluções projetadas para aplicações de alta performance, que costumam executar em um mesmo domínio administrativo.

2.2.3 EBMs e MOMs

EBMs (*Event-Based Middlewares*) e MOMs (*Message-Oriented Middlewares*), por sua vez, propiciam um desacoplamento entre as entidades através do modelo *publish/subscribe*, fazendo com que a comunicação seja de maneira indireta. O modelo *publish/subscribe* permite que um evento seja publicado e consumido por quantas entidades estiverem interessadas, de forma que é possível realizar comunicação em grupo. Uma vez que a publicação e con-

sumo de eventos usam primitivas análogas a *send* e *receive*, a integração com a linguagem de programação não é tão boa quanto em objetos distribuídos. EBMs e MOMs costumam ser incrementados através de serviços como *roteamento de mensagens*, *store-and-forward* e *gerenciamento de transações*, que são feitos por entidades intermediárias, chamadas servidores ou *containers*. Logo, a presença do mecanismo de *store-and-forward* elimina a dependência de tempo entre as entidades que se comunicam. No que diz respeito ao uso do modelo *event-driven*, os servidores usados por EBMs e MOMs em geral o usam, repassando eventos para as aplicações, que também podem usar este mesmo modelo para consumir estes eventos.

Existem duas abordagens comuns em EBMs e MOMs no que diz respeito a falhas. A primeira delas consiste em não realizar qualquer tratamento, deixando-o para o programador da aplicação [BGT+01]. A desvantagem desta abordagem está no fato de que o programador precisa se desviar da lógica de negócio para lidar com funcionalidades que poderiam ser oferecidas pela infra-estrutura de comunicação, uma vez que detecção de falhas é um aspecto essencial em aplicações distribuídas. A segunda abordagem consiste em considerar um modelo *crash-recovery*, no qual uma entidade que falha sempre se recupera. As abordagens que consideram este modelo costumam fornecer suporte para mensagens *offline*. Dessa Forma, todas as mensagens endereçadas a uma entidade que falhou serão armazenadas no servidor (ou serviço de eventos) de maneira persistente (e.g., em um banco de dados), de forma que, quando a entidade retornar, receberá todas as mensagens pendentes. Contudo, existem aplicações que precisam reagir a falhas de maneira diferente, ou seja, não podem simplesmente esperar que a entidade se recupere e volte ao sistema. Para estas aplicações, resta apenas a primeira abordagem.

Em relação à comunicação na presença de *firewalls* e NATs, EBMs e MOMs podem tirar vantagem de sua arquitetura. Uma vez que as mensagens sempre passam por uma entidade intermediária, em geral um servidor, caso exista um servidor por domínio administrativo, pode-se usá-lo como ponto único de acesso para todos os objetos que estão em um determinado domínio. Sendo assim, é preciso abrir apenas uma porta no *firewall* (ou criar tradução reversa para um único endereço no NAT) para permitir que todas as aplicações em um domínio se comuniquem com outros domínios.

2.2.4 Twisted

Twisted [Fet06] é um *framework* construído em Python [Fou06] sobre um modelo de programação *event-driven*, que permite comunicação através de uma enorme gama de protocolos (e.g., TCP, UDP, SSH, *sockets*). Devido a esta possibilidade de escolha de protocolos, Twisted permite comunicação através de troca de mensagens (através de protocolos que usam primitivas semelhantes a *send* e *receive*), de forma que a comunicação pode ser pouco integrada à linguagem de programação. Porém, como veremos na Seção 2.3.3, Twisted também oferece comunicação assíncrona bem integrada à linguagem de programação. Dependendo do protocolo de comunicação escolhido ou implementado, Twisted pode permitir endereçamento direto ou *publish/subscribe*, fazer *unicast* ou *multicast* e ainda optar pela não dependência de tempo caso realize comunicação semelhante a MOMs ou EBMs.

Aspectos relacionados à comunicação assíncrona bem integrada à linguagem serão apresentados na Seção 2.3.3, onde também apresentaremos detalhes sobre detecção de falhas e comunicação na presença de *firewalls* e NATs no Twisted.

2.3 Objetos Distribuídos Assíncronos

Existem soluções que buscam unir a boa integração com a linguagem, observada no modelo de objetos distribuídos, com o aspecto não-bloqueante de chamadas assíncronas. As abordagens variam desde adaptações de middlewares já existentes, como ARMI, até soluções mais sofisticadas, como o Twisted.

2.3.1 ARMI

Existem soluções denominadas ARMI (*Asynchronous Remote Method Invocation* - Invocação Remota de Métodos Assíncronos) [RWB97; FCO99] que estendem o modelo usado por Java RMI para prover comunicação assíncrona. No entanto, tais soluções consistem em criar threads extras automaticamente dentro dos *stubs* de comunicação, sendo uma thread extra para cada nova invocação remota. Esta thread bloqueia na invocação remota e libera a thread da aplicação para continuar seu processamento local. Apesar da simplicidade e boa integração com a linguagem, estas soluções têm as mesmas desvantagens de Java RMI, uma

vez que usam a mesma camada de comunicação, além de proporcionar a possibilidade de explosão de threads no lado cliente quando várias invocações simultâneas são realizadas. O retorno da chamada assíncrona é armazenado em um objeto *future* [RHH85], de forma que a thread da aplicação pode consultar este objeto para recuperar o resultado de uma invocação (podendo bloquear ao realizar a consulta, caso o resultado não esteja disponível). Note que o estilo de programação usado por Java RMI não é totalmente preservado, uma vez que a aplicação precisa recuperar resultados através de *futures*. Além disso, o mecanismo de detecção de falhas padrão de Java RMI deve ser incrementado, pois a thread da aplicação deve ser notificada sobre falhas, uma vez que a thread que realiza detecção é a nova thread criada dentro dos *stubs*, que bloqueia até receber a resposta ou detectar uma falha.

2.3.2 Comunicação Assíncrona em CORBA

Além da comunicação síncrona do modelo de objetos distribuídos, CORBA inicialmente fornecia dois modelos de comunicação não-bloqueante: (i) chamadas em direção única (*oneway calls*) e (ii) chamadas síncronas adiadas (*deferred synchronous calls*). O modelo *oneway* é formado por invocações que representam requisições sem retorno, de forma que a especificação de CORBA define a semântica para este tipo de invocações como de "melhor esforço", ou seja, ORBs não precisam lançar um erro caso a invocação não possa ser executada remotamente. Schmidt *et al.* [SV99] mostram que *oneway calls* não garantem a semântica não-bloqueante nem entrega garantida. A idéia dos criadores consistia em permitir a implementação deste modelo através de protocolos de transporte não confiáveis. No modelo *deferred synchronous*, após realizar uma requisição, o cliente pode verificar através de *polling* se a resposta referente a esta requisição foi retornada (assim como *futures* [RHH85] em ARMI) ou realizar uma chamada bloqueante em separado para esperar por uma resposta. Este modelo é ineficiente devido ao fato de usar a Interface de Invocação Dinâmica (DII - *Dynamic Invocation Interface*), que aloca memória e copia dados excessivamente [SV99].

Para endereçar os problemas presentes em *oneway calls* e *deferred synchronous calls*, a especificação de troca de mensagens através de CORBA [OMG98] introduziu o modelo de invocação assíncrona de métodos (AMI - *Asynchronous Method Invocation*), que especifica dois modelos não-bloqueantes similares aos apresentados acima. No modelo de *Polling*, a invocação retorna um objeto `Poller`. O cliente pode usar os métodos do `Poller` para obter

o estado de uma requisição e o valor retornado pelo objeto invocado (de maneira bloqueante ou não-bloqueante). No modelo de *callback*, uma entidade chamada `ReplyHandler` é passada como parâmetro quando o cliente invoca um objeto remoto, de forma que o retorno deste objeto remoto é automaticamente redirecionado para o `ReplyHandler` pelo ORB. Entretanto, poucos ORBs implementam esta especificação completamente.

2.3.3 Comunicação Assíncrona no Twisted

Através do Twisted (*framework* construído em Python sobre um modelo de programação *event-driven*), é possível realizar comunicação assíncrona de maneira bem integrada à linguagem de programação, sem precisar lidar com a complexidade do modelo de threads. Apesar de oferecer classes de alto nível para que o programador desenvolva aplicações distribuídas usando o protocolo de sua escolha, Twisted é flexível o suficiente para permitir que o programador configure a maneira como um determinado protocolo funciona ou até mesmo implemente seu próprio protocolo. Uma vez que é possível construir aplicações sobre vários protocolos, Twisted oferece também a possibilidade de integrar aplicações em um mesmo processo, ou seja, compartilhar dados entre aplicações construídas sobre protocolos diferentes.

O componente principal da arquitetura *event-driven* do Twisted é o *event-loop*. Conforme explicado na Seção 1.1, trata-se de uma função que roda indefinidamente esperando por eventos em uma fila. Quando um evento chega, o *event-loop* dispara uma função *event-handler* para tratar o evento.

Invocações com retorno são gerenciadas assincronamente através do uso de componentes especiais, chamados `deferreds`. Ao invocar uma função que realiza uma execução remota de maneira assíncrona, esta função retorna imediatamente um objeto do tipo `Deferred`. Um `Deferred` tem comportamento baseado em *futures* [RHH85] e *promises* [LS88]. Um *future* é um objeto vazio, localizado no mesmo espaço de endereçamento do cliente, e que está associado a uma invocação remota assíncrona realizada por este cliente. Quando o processamento remoto é concluído, o objeto remoto armazena o resultado da invocação no *future*, permitindo que o cliente recupere o resultado quando for necessário, através de consulta ao *future*. *Promises* são mais elaborados que *futures*, permitindo, por exemplo, que exceções do lado remoto sejam propagadas de volta para o cliente. No Twisted, é permitido

adicionar funções como *callback* e/ou *errorback* em um `Deferred`. Dessa forma, caso a invocação remota seja bem sucedida, o método de *callback* associado ao `Deferred` desta invocação será executado no lado cliente, de forma que este *callback* poderá reagir corretamente diante do resultado da invocação. Caso a invocação remota retorne uma exceção, o método de *errorback* será executado no cliente, tratando o erro da maneira mais adequada. Note que, erros de comunicação podem ser devolvidos como exceções via *errorback*. Isso significa que o Twisted pode usar os *timeouts* da camada de comunicação para detectar falhas de componentes e repassar tais falhas através de exceções, usando o método de *errorback* associado a uma determinada invocação.

Em relação à comunicação na presença de *firewalls* e NATs, Twisted também é dependente do protocolo escolhido. Caso o protocolo seja TCP, objetos distribuídos assíncronos através de Twisted sofrem os mesmos problemas enfrentados por Java RMI em relação a *firewalls* e NATs. Porém, é possível implementar novos protocolos sobre Twisted que possibilitem comunicação através de uma entidade intermediária, i.e., um *relay*, que se torna o único ponto de acesso a um domínio, de forma que apenas uma porta precisa estar aberta no *firewall* deste domínio.

Conforme veremos no Capítulo 3, Twisted é a solução mais parecida com o JIC, apesar de algumas diferenças.

2.4 Outras Preocupações Importantes em Sistemas Distribuídos

Como vimos no Capítulo 1, evitar problemas relacionados à concorrência, manter boa integração com a linguagem de programação e possibilitar comunicação não-bloqueante é um desafio, especialmente pelo fato de que os objetos comunicantes estão localizados em espaços de endereçamento distintos, possivelmente em máquinas distintas. Outros aspectos relevantes para esta dissertação são *detecção de falhas parciais* e *comunicação na presença de firewalls e NATs*, que já foram apresentados nas seções anteriores quando exploramos as características das soluções existentes. Em relação a estes dois últimos aspectos, apresentaremos adiante algumas soluções que fazem parte do estado da arte e que consideramos relevantes, mas não são usadas pelas soluções até então descritas.

2.4.1 Detecção de Falhas

Um dos maiores desafios ao lidar com sistemas distribuídos consiste em saber se uma entidade falhou ou se apenas está respondendo muito lentamente devido a um congestionamento na rede ou uma grande carga no sistema. Uma vez que na maioria dos sistemas não existe limite superior de tempo para uma mensagem alcançar o seu destino, dizemos que tais sistemas são inerentemente assíncronos[1]. Entretanto, uma entidade não pode esperar por uma resposta de outra eternamente, caso esta outra tenha falhado. Por isso, é necessário inserir um certo grau de sincronia no sistema. Em geral, isto é feito através da implementação de um mecanismo de detecção de falhas. Chandra e Toueg [CT96] mostraram que é possível construir sistemas distribuídos confiáveis que podem lidar com falhas mesmo que os detectores de falhas não sejam perfeitos, i.e., que possam cometer erros. Tal abordagem consiste em um mecanismo de detecção de falhas distribuído, de forma que existe um módulo de detecção de falhas acoplado a cada entidade do sistema.

Estes detectores de falhas "não confiáveis" têm sido amplamente usados como elementos essenciais em sistemas distribuídos. O monitoramento realizado por tais detectores baseia-se no modelo de *heartbeats*, que são mensagens enviadas pela entidade monitorada, informando que esta entidade está operacional.

Na intenção de melhor abstrair os detalhes relacionados à detecção de falhas, existem trabalhos que consideram detectores de falhas como objetos de primeira ordem em um sistema distribuído [FDGO99], i.e., toda a complexidade inerente à detecção de falhas é encapsulada em interfaces bem definidas. Tais interfaces abstraem o comportamento de detecção de falhas que deve ser conhecido pela aplicação. Implementar este modelo é um passo em direção a fornecer flexibilidade para o controle de qualidade de serviço para a detecção de falhas. O *framework* descrito em [FDGO99], descreve basicamente três tipos fundamentais:

1. `Monitor`: objeto responsável por monitorar o estado de outros objetos, ou seja, verificar se um objeto falhou ou está no ar;

[1] Apesar de estarmos usando o termo "assíncrono" com o significado "não-bloqueante", no contexto deste parágrafo, sistemas assíncronos são sistemas em que não há limite superior conhecido para o tempo de transmissão de uma mensagem. De maneira análoga, em um sistema síncrono, existe um limite superior para o tempo de transmissão de uma mensagem.

2. `Monitorável`: objeto monitorado pelo `Monitor`;

3. `Notificável`: objeto interessado junto ao `Monitor` no estado de um `Monitorável`.

Considerando estes três tipos, a detecção de falhas pode acontecer através de dois modelos:

1. **modelo *push*:** neste modelo, apresentado na Figura 2.2, cada `Monitorável` periodicamente envia um *heartbeat* para o `Monitor`. O período entre envio de dois *heartbeats* é chamado *intervalo entre heartbeats*, representado por Δ_i. O `Monitor` mantém uma expectativa em relação ao tempo de chegada do próximo *heartbeat*. Se o *heartbeat* não chega dentro do tempo esperado Δ_{to} (com $\Delta_{to} > \Delta_i$), o detector de falhas do `Monitor` considera que o `Monitorável` falhou, ou seja, suspeita do `Monitorável`. A cada mudança de estado (suspeita de falha ou aparecimento) percebida pelo `Monitor`, o `Notificável` é informado sobre tal mudança.

Figura 2.2: Modelo de monitoramento *push*

2. **modelo *pull*:** neste modelo, apresentado na Figura 2.3, a informação flui no sentido oposto: entidades interessadas só recebem informações quando as requisitam. O `Monitor` periodicamente solicita informação sobre o estado de um `Monitorável`. O período entre solicitações de *heartbeats* também é chamado representado por Δ_i. Caso o `Monitorável` não responda dentro do tempo esperado Δ_{to} (com $\Delta_{to} > \Delta_i$), o detector de falhas do `Monitor` suspeita do `Monitorável`, ou seja, considera que o `Monitorável` falhou. Da mesma forma, o `Notificável` requisita informações sobre o estado de um objeto ao `Monitor`, que responde com a sua informação mais atual.

Apesar de ser menos eficiente (devido à troca mensagens em duas vias), a abordagem *pull* é mais flexível, pois a entidade monitorada não precisa ter a noção do intervalo entre

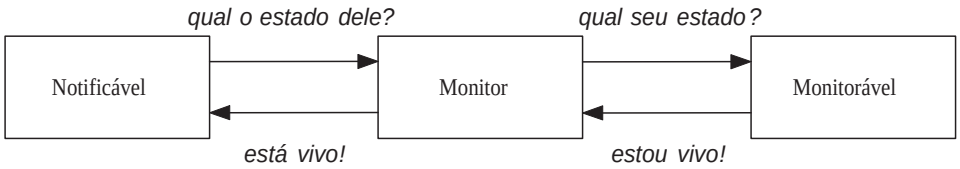


Figura 2.3: Modelo de monitoramento *pull*

heartbeats (Δ_i). Quem controla este valor é o `Monitor`, que pergunta sobre o estado do `Monitorável` de tempos em tempos. Logo, o modelo *pull* é mais apropriado para abordagens de detecção adaptativas (explicadas mais adiante), em que o Δ_i pode ser modificado em tempo de execução para se adaptar à qualidade de serviço definida pela aplicação. Abordagens adaptativas que usam um modelo *push* modificam apenas o timeout (Δ_{to}) em tempo de execução [CBO05].

Através dos modelos de detecção acima, muitos trabalhos visam prover qualidade de serviço para a detecção de falhas em uma aplicação. Tal qualidade de serviço é refletida basicamente em duas métricas [CTA02]:

1. **Tempo de detecção (T_D):** tempo decorrido entre a falha de uma entidade p e o momento em que outra entidade q (interessada na falha de p) passa a suspeitar de p permanentemente;

2. **Taxa média de erros (λ_M):** fornece a medida da taxa em que o detector de falhas suspeita erradamente de outras entidades. Por isso, λ_M é também conhecido como *taxa de falsas suspeições*.

Se acontece um aumento na carga do sistema em que uma entidade remota executa, ou há congestionamento na rede, o detector de falhas de uma entidade interessada pode considerar que a entidade remota falhou, pois o próximo *heartbeat* não chegou dentro do limite Δ_{to} esperado. Logo depois, quando o *heartbeat* chega de maneira atrasada, o detector de falhas percebe que cometeu uma falsa suspeição. Diante disso, percebe-se que ajustar as duas métricas acima é um *trade-off*: na medida em que T_D diminui, λ_M aumenta. da mesma forma, na medida em que T_D aumenta, λ_M diminui.

O balanceamento do tempo de detecção e da taxa de falsas suspeições recebe influência direta do ambiente. Uma configuração do Δ_{to} que oferece uma boa qualidade de serviço em

uma rede local pode não fornecer uma boa qualidade de serviço em uma rede de longa distância. Da mesma forma, variações de carga nas entidades e congestionamentos na rede podem degradar a qualidade de serviço do mecanismo de detecção se o Δ_{to} permanece inalterado. Além disso, diferentes aplicações possuem diferentes requisitos quanto à detecção de falhas. Portanto, na intenção de se adaptar às mudanças de ambiente e aos diferentes requisitos de aplicações quanto à detecção de falhas, foram criados detectores de falhas adaptativos, os quais modificam o valor de Δ_{to} dinamicamente, de acordo com as variações no ambiente. As duas soluções adaptativas mais conhecidas são descritas abaixo:

1. **Detector de falhas de Chen:** Chen et al. [CTA02] propõe um detector de falhas baseado na análise probabilística do tráfego na rede. O protocolo usa os tempos de chegada dos *heartbeats* recentes para estimar o tempo de chegada do próximo *heartbeat*. O *timeout* Δ_{to} é configurado de acordo com a estimativa e uma margem de segurança (α) é adicionada a este *timeout*. A cada chegada de *heartbeat*, a estimativa do tempo de chegada do próximo *heartbeat* é computada, mas a margem de segurança é computada apenas uma vez, baseada em parâmetros de qualidade de serviço.

2. **Detector de falhas de Bertier:** Bertier et al. [BMS02] propõe um detector de falhas adaptativo baseado no detector de falhas de Chen, mas a margem de segurança α é calculada dinamicamente, através do método de estimativa de Jacobson para o *round-trip time* [Jac95].

Dentre as soluções adaptativa, existem trabalhos que, além de lidar com detecção de falhas probabilística, associam níveis de confiança aos `Monitoráveis` (e.g., o detector de falhas φ *Accrual* [HDYK04]) ao invés de apenas considerarem se as entidades monitoradas falharam ou não.

No modelo φ *Accrual*, o `Monitor` solicita informações sobre o `Monitorável` seguindo o tempo Δ_i e armazena os tempos de chegada das respostas em uma janela de tamanho n. Assumindo que a amostra representa uma população com distribuição normal e baseando-se nos valores observados nesta janela, o detector de falhas é capaz de associar valores de *confiança na falha* de um objeto em um dado momento. Basicamente, através da amostra, o detector de falhas é capaz de calcular o tempo provável de chegada do próximo

heartbeat. Na medida em que o tempo passa e o próximo *heartbeat* não é recebido, a confiança na falha do objeto aumenta. Note que, dessa forma, o estilo de detecção de falhas deixa de ser binário (no qual o `Monitor` suspeita ou confia) e passa a associar níveis de confiança na falha de um objeto. Valores pequenos para este nível indicam que o objeto tem grande probabilidade de estar operacional. À medida que a confiança na falha aumenta, maior é a probabilidade de que o `Monitorável` tenha realmente falhado. Diante disso, o `Monitor` calcula o nível de confiança na falha de um `Monitorável`, e o `Notificável` solicita informações ao *Monitor*, estabelecendo um limite superior, acima do qual considera que um objeto falhou. Quando o nível de confiança observado pelo `Monitor` ultrapassa este limite, o `Notificável` é informado sobre a suspeita do `Monitorável`.

2.4.2 Comunicação na presença *Firewalls* e NATs

Redes privadas (implementadas através de NATs - *Network Address Translators*) surgiram como uma solução para os problemas decorrentes do espaço de endereçamento insuficiente do IPv4 [TS06; CD05]. Porém, redes privadas também podem ser usadas para incrementar a segurança de um domínio, escondendo máquinas que em princípio não têm necessidade de estar publicamente acessíveis. Cada máquina em uma rede privada possui um IP privado, que não pode ser acessado diretamente por máquinas externas ao domínio delimitado pelo NAT. O NAT associa o seu IP público a cada máquina da rede privada e roteia pacotes recebidos que são endereçados a seu IP público para as respectivas máquinas internas. Em geral, o hardware que executa o NAT é o mesmo que executa o *firewall*, solução que endereça exclusivamente aspectos de segurança. *Firewalls* são barreiras de proteção que regulam o tráfego de rede entre redes distintas, não permitindo acesso não autorizado às mesmas.

Mesmo com o desenvolvimento do IPv6, redes privadas tendem a continuar existindo. Embora IPv6 resolva o problema do espaço de endereçamento do IPv4, ainda existem questões de segurança que redes privadas endereçam. Além disso, redes privadas são de fácil manutenção e também endereçam problemas econômicos [Ram01], uma vez que o administrador não precisa pagar por um IP público para cada máquina em sua rede.

Diante deste contexto, existe a necessidade de desenvolver maneiras de lidar com a comunicação na presença de *firewalls* e NATs, uma vez que seus comportamentos limitam a conectividade. Adiante, apresentaremos três alternativas: *hole punching*, conexão reversa e

o uso de relays. O comportamento comum de *firewalls* e NATs consiste em bloquear tentativas de início de conexão originadas de máquinas externas ao domínio. Tráfego externo é apenas permitido se a conexão foi iniciada por uma entidade interna ao domínio delimitado por *firewalls* e NATs. Sendo assim, o mecanismo padrão de negociação de conexão TCP não funciona caso a tentativa de iniciar a conexão seja originada de uma máquina externa ao domínio, como mostra a Figura 2.4:

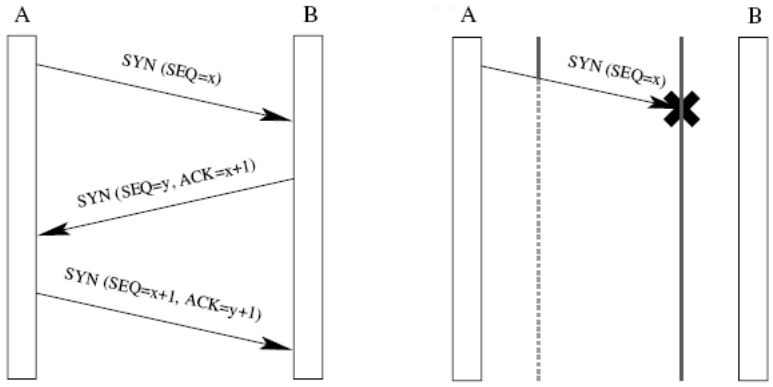


Figura 2.4: Negociação de conexão padrão do TCP à esquerda. Na direita, temos a situação em que existe um *firewall* em ambas as máquinas.

Além da negociação padrão, TCP também define "iniciação simultânea" de conexão (*TCP Splicing*) [IET06d], uma maneira simétrica de estabelecer conexão. Isto acontece quando duas máquinas tentam simultaneamente (ou quase simultaneamente) iniciar a conexão. Neste caso, ambos os *firewalls* enxergam a conexão como tráfego iniciado internamente e permitem a comunicação, como ilustrado na Figura 2.5:

Esta possibilidade de conexão simétrica é explorada através de um mecanismo chamado *hole-punching* [For05]. Trata-se de uma técnica que visa aproveitar o comportamento padrão de *firewalls* e NATs. Não existe uma especificação que dita como NATs e *firewalls* devem se comportar, mas a IETF iniciou um grupo chamado BEHAVE, que define as melhores práticas para o comportamento de NATs. A idéia deste grupo consiste em fazer NATs e *firewalls* bem adaptados a sistemas *peer-to-peer*.

Hole-punching é uma técnica que faz uso de uma entidade intermediária, chamada *relay*. Esta entidade participa apenas no processo de iniciar a conexão. Depois de iniciada a conexão, a comunicação acontece diretamente entre as duas partes envolvidas. A idéia deste mecanismo é bastante simples e aproveita um comportamento muito comum em *fi-*

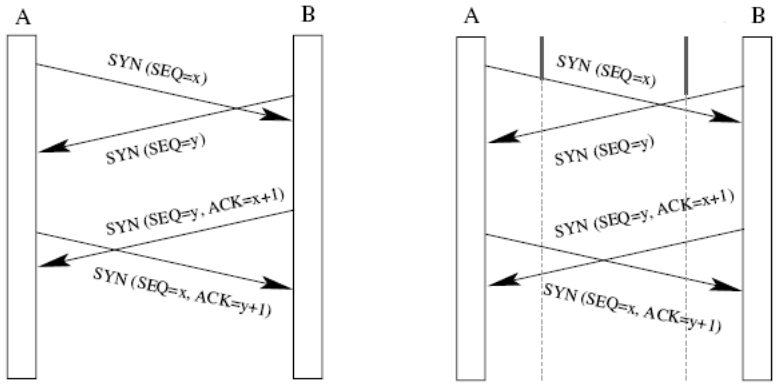


Figura 2.5: Iniciação simultânea de conexão TCP à esquerda. Na direita, temos a situação em que existe um *firewall* em ambas as máquinas.

rewalls: quando uma máquina A interna ao *firewall* tenta iniciar uma conexão com uma máquina B externa ao *firewall*, o *firewall* mantém esta informação e "abre um buraco" em si mesmo. "Abrir um buraco" significa passar a aceitar conexões iniciadas por B, uma vez que A pretende se comunicar com tal máquina. A partir deste momento, uma tentativa de negociar conexão iniciada por B é vista como uma resposta ao processo iniciado por A. O mecanismo funciona da seguinte maneira:

Suponha que A e B estão em redes privadas diferentes. Cada um possui um endereço público e outro privado, definidos pelo NAT. Inicialmente, ambos se conectam a um *relay* R e informam seus endereços público e privado. Neste momento, R possui um mapeamento que associa as identificações de A e B aos seus respectivos IPs públicos e privados.

Logo depois, suponha que A deseja se conectar a B. O primeiro passo consiste em solicitar a R ajuda para se conectar a B. O *relay* R então fornece a A os IPs público e privado de B, fazendo o mesmo para B em relação a A.

Em seguida, ambos tentam iniciar conexão com o outro através dos IPs públicos e privados recebidos, parando este processo quando a conexão for estabelecida (se ambas as máquinas estiverem em uma mesma rede privada, muito possívelmente a conexão será estabelecida através dos IPs privados). Uma vez que ambos tentaram iniciar uma conexão, eles "abriram um buraco" em seus *firewalls*, permitindo que a conexão seja estabelecida.

Hole-punching é uma generalização de uma técnica conhecida como *conexão reversa* (*connection reversal*), que funciona apenas quando uma máquina se encontra em uma rede

privada ou protegida por *firewalls* e a outra está acessível em uma rede pública. Através desta técnica, a máquina A usa o *relay* R para informar a necessidade de se comunicar com B. R então contacta B e solicita que B inicie conexão com A, que não está por trás de *firewall* ou NAT. Logo, a limitação desta técnica está no fato de não suportar comunicação *NAT-to-NAT*.

Uma outra técnica, bastante usada (em JXTA [JXT06], por exemplo), consiste em usar *relays* para realizar toda a comunicação. Se uma máquina A precisa se comunicar com uma máquina B, A envia mensagens para um *relay* (conhecido a priori). O *relay* armazena as mensagens recebidas, e entrega-as em algum momento no futuro, quando B solicitar ao *relay* as mensagens endereçadas a si.

Capítulo 3

Definição do Problema

No Capítulo 1, apresentamos superficialmente os problemas relacionados à programação concorrente devido ao uso do modelo de threads. Destacamos a complexidade extra imposta por este modelo em aplicações concorrentes que executam em sistemas distribuídos, quando comparadas a aplicações concorrentes locais. Tal apresentação, apesar de não explorar os problemas em detalhes, serviu para contextualizar a tese investigada neste trabalho. Neste capítulo, iremos nos aprofundar em tais problemas, definindo-os de maneira pormenorizada e dando suporte à motivação apresentada no primeiro capítulo.

3.1 O Problema com Threads

Apesar de o modelo de threads ter se tornado a abordagem predominante para programação concorrente, podemos identificar dois aspectos que vão de encontro ao uso de threads como modelo ideal para este tipo de programação: (i) a dificuldade de raciocinar em relação a códigos multi-threaded, dada a complexidade de programação concorrente e o não-determinismo do modelo de threads [Her05; Edw06] e (ii) o problema de *locks* não serem componentizáveis [Her05]. Tais aspectos tornam-se mais relevantes na medida em que arquiteturas de computadores paralelas passam a ser cada vez mais comuns, pois estas arquiteturas conseguem explorar melhor o paralelismo das aplicações concorrentes, fazendo com que *bugs* relacionados à programação concorrente apareçam com mais frequência.

Primeiramente, temos que programação concorrente é demonstravelmente mais complexa que programação sequencial. Por exemplo, existem técnicas fundamentais para análise

estática de programas sequenciais [BPS00; ECCH00; EGHT94], que verificam se o comportamento de um programa está de acordo com o esperado. Uma delas, a análise interprocedural sensível ao contexto (*context-sensitive analysis*) [RHS95; SP81], leva em consideração o contexto da chamada ao analisar a invocação de uma função. Para programas concorrentes, tal análise não é suficiente, sendo necessária uma análise sensível à sincronização. Entretanto, realizar simultaneamente estes dois tipos de análise é um problema comprovadamente indecidível [Gen00]. Além disso, programação concorrente requer que os programadores raciocinem de uma maneira que não é fácil para seres humanos [Her05]. Além desta complexidade, o modelo de threads impõe mais dificuldades devido ao seu não-determinismo. Desenvolver código multi-threaded significa assumir um modelo não-determinístico e tentar limitar esse não-determinismo através de mecanismos como semáforos, monitores e *locks* [And01]. Em [Edw06], sugere-se que o paradigma ideal para programação concorrente deve ser determinístico, de forma que o não-determinismo deve ser inserido apenas onde necessário. Isso evitaria as complicações decorrentes do modelo de threads, em que *bugs* são muito frequentes [HP04].

O segundo problema trata do fato de *locks* não serem componentizáveis. Quando duas ou mais threads executam simultaneamente o mesmo trecho de código em um componente, podendo modificar estado, é preciso usar algum mecanismo que evite condições de corrida [And01]. *Locks* são os mecanismos mais simples e mais usados para este fim. Para ter acesso à região crítica, cada thread precisa adquirir o *lock* referente àquela região, executar a computação necessária e, apenas depois, liberar o *lock*. Contudo, não se pode combinar dois componentes que usam *locks* corretamente e saber se o resultado desta combinação está correto sem analisar ambos os códigos [Her05]. Isto se deve ao fato de que não se pode ter certeza da ordem em que os *locks* são adquiridos em um componente sem verificar seu código. Caso a ordem de aquisição dos *locks* gere uma situação em que se tem espera circular, essa combinação pode facilmente ocasionar *deadlocks* [And01]. Note que este problema vai contra uma técnica muito importante na engenharia de software, a modularização, na qual componentes são combinados sem necessitar conhecer suas implementações, possibilitando reuso de componentes e a construção de componentes maiores. Em programas concorrentes, todavia, não se pode combinar dois componentes baseados em *locks* sem conhecer suas *implementações*.

3.2 Threads e Objetos Distribuídos

Devido ao uso do modelo de threads e à necessidade de desenvolver código multi-threaded para objetos remotos, o paradigma de objetos distribuídos herda todos os problemas inerentes ao modelo de threads. Porém, existem pelo menos três agravantes quando consideramos o desenvolvimento de aplicações sobre objetos distribuídos. Como veremos adiante, (i) o problema de componentização de *locks* em objetos distribuídos é ainda mais grave do que o problema mencionado na Seção 3.1. Também veremos que (ii) raciocinar sobre o modelo de threads aplicado a objetos distribuídos é mais difícil quando comparado a aplicações concorrentes em um mesmo processo e que (iii) o suporte para depuração de aplicações multi-threaded em sistemas locais não funciona da mesma forma para sistemas distribuídos.

Para contextualizar o agravante no problema de componentização de *locks*, considere o exemplo apresentado no trecho de código em Java da Figura 3.1. Considere que os códigos de `A` e `B` são adaptados para usar objetos distribuídos. Nesse cenário, um objeto `A` invoca um método remoto em um objeto `B`, cuja execução implica na chamada de um método de *callback* em `A` antes que o resultado seja retornado. Assim descrito, este cenário será executado corretamente.

```
public class A {

   B objeto_B;
   ...
   public void metodo_A(){
      Object retorno = objeto_B.metodo_B( this );
   }

   public void callback() {
   ...
   }

}
```

(a)

```
public class B {

   public Object metodo_B(A objeto_A) {
      objeto_A.callback();
      Object retorno = new Object();
      return retorno;
   }

}
```

(b)

Figura 3.1: Código de duas aplicações `A` e `B`, respectivamente nas Figuras (a) e (b)

Considere agora a modificação no código de `A` apresentada no trecho de código da Figura 3.2. Nesse caso, o objeto `A` adquiriu um determinado *lock* antes de executar a invocação remota. Note que a invocação do método de *callback* realizada por `B` em `A` requer a posse do *lock* para que a execução prossiga. Se ambos os objetos estivessem no mesmo espaço

de endereçamento, a thread que invoca o objeto `B` seria a mesma que executa o método de *callback*, que já tem a posse do *lock*. No entanto, se os objetos estiverem em processos diferentes, se comunicando através do paradigma de objetos distribuídos, o comportamento não será o mesmo. Uma vez que cada chamada remota implica na criação de uma nova thread no lado chamado, a thread que realiza o *callback* irá bloquear, pois não possui o *lock*. Mas a thread que o possui também está bloqueada, esperando por uma resposta que depende da execução do método de *callback*. Isso significa que temos uma situação de espera circular, caracterizando um *deadlock* distribuído.

```
public class A {

   B objeto_B;
   Object lock;
   ...
   public void metodo_A(){
      synchronized (lock) {
         Object retorno = objeto_B.metodo_B( this );
      }
   }

   public void callback() {
      synchronized (lock) {
       ...
      }
   }

}
```

Figura 3.2: Código de `A` modificado com bloco *synchronized* em suas chamadas

Temos portanto uma nova situação em que o problema de locks não componentizáveis se manifesta nos sistemas baseados em objetos distribuídos.

De fato, o problema com threads é ainda mais grave ao usar objetos distribuídos porque o raciocínio sobre threads se torna mais complicado. Isso se deve às características inerentes aos sistemas baseados em objetos distribuídos. Primeiramente, objetos distribuídos não implementam o modelo orientado a objetos perfeitamente. Em geral, na passagem de parâmetros em métodos remotos, apenas objetos remotos são passados por referência, sendo os demais objetos passados por cópia. Logo, objetos que não são remotos não podem ser compartilhados entre vários espaços de endereçamento. Cada processo possui sua própria cópia para cada objeto passado como parâmetro. Consequentemente, um mesmo código pode se comportar de maneira diferente ao executar em um único processo e ao executar em espaços

de endereçamento diferentes. Considere o trecho de código apresentado na Figura 3.3:

```
public class C {

   public void metodo_C( Object objeto ){
      synchronized (objeto) {
         ...
      }
   }

}
```

Figura 3.3: Código da aplicação `C`, cujo método `metodo_C` realiza controle de concorrência.

Se duas instâncias da classe `C` em um mesmo processo recebem chamadas simultâneas ao método `metodo_C` originadas por threads diferentes, recebendo o mesmo parâmetro, haverá controle de exclusão mútua. Apenas uma das threads por vez poderá adquirir o *lock* e entrar na região crítica. No entanto, se cada uma das instâncias estiver em um processo diferente, ambas recebendo o mesmo objeto como parâmetro da chamada a `metodo_C`, cada instância criará sua própria cópia do parâmetro recebido, ou seja, não há memória compartilhada.

Caso seja necessário haver memória compartilhada, o objeto passado como parâmetro deverá ser um objeto remoto. Dessa forma, é possível passá-lo como referência. As invocações realizadas neste objeto poderão ter controle de sincronização realizado no próprio objeto. Porém, as primitivas de sincronização das linguagens de programação não valem para objetos remotos. Criar um bloco *synchronized* para um objeto remoto significa adquirir o *lock* do objeto local que o representa, ou seja, de seu *stub*.

Por fim, depurar código concorrente distribuído é bem mais difícil do que depurar código concorrente que roda em um mesmo processo. Tal dificuldade se deve ao fato de que, ao realizar uma invocação remota, a continuação da thread local é feita por uma nova thread no lado remoto. Acompanhar a execução de uma invocação remota significa depurar código em processos diferentes (possivelmente em máquinas diferentes) e mapear threads locais às suas continuações em outros espaços de endereçamento. Em geral, a depuração de sistemas multi-threaded é feita com a ajuda das informações sobre as pilhas de execução de cada thread. Tais informações são conhecidas como *thread dumps*. Contudo, ao programar através de objetos distribuídos, *thread dumps* não são muito úteis para depurar possíveis *deadlocks* distribuídos, pois cada *thread dump* armazena informações apenas sobre as threads locais,

ou seja, não existe informação sobre a continuação de uma thread local que está bloqueada. Em ambientes locais, *deadlocks* são facilmente detectados através do *thread dump*.

3.3 Objetos Distribuídos e Aplicações não Bloqueantes

É importante salientar que existe uma outra limitação quanto ao uso de objetos distribuídos além das impostas pelo modelo de threads. Devido à sua natureza bloqueante, o paradigma de objetos distribuídos não é uma solução apropriada para aplicações em que o cliente tem outras coisas a fazer além de bloquear à espera de uma resposta do servidor (e.g., aplicações interativas e aplicações paralelas). A maneira mais óbvia de contornar este problema consiste em criar uma nova thread no lado cliente para bloquear em cada invocação, liberando a thread original da aplicação para continuar realizando processamento.

Entretanto, esta abordagem possui pelo menos duas desvantagens. Primeiramente, criar uma nova thread para cada nova invocação remota pode ocasionar uma explosão de threads no lado cliente, caso esteja executando um grande número de invocações remotas simultâneas. O desenvolvedor precisaria, portanto, implementar um mecanismo explícito de gerência de threads (e.g., *thread pools*). Além disso, uma vez que a thread original da aplicação não bloqueia nas invocações remotas, o mecanismo de detecção de falhas deve ser melhorado para notificar a aplicação em caso de falhas. Note que a detecção de falhas continua acoplada à invocação que, por sua vez, está sendo executada por uma nova thread. Na ocorrência de uma falha, a thread que receberá a exceção será uma thread que está bloqueada. Logo, é preciso que exista um mecanismo que notifique a thread original quando uma falha acontece. Um mecanismo análogo também deve ser usado para entregar o retorno de uma invocação à thread original, uma vez que o retorno também é recebido pela nova thread.

Outra possível solução consiste em fazer com que cada invocação remota apenas crie uma requisição no lado servidor para ser processada posteriormente, de forma que o cliente pode desbloquear de maneira rápida. Após processar a requisição, o servidor pode retornar o resultado para o cliente através da invocação de um método de *callback*. Todavia, para que o servidor possa invocar um método de *callback* no cliente, deve ser possível iniciar conexão em ambas as direções (do servidor para o cliente e vice-versa), o que é um problema devido ao crescente uso de *firewalls* e NATs. Além disso, o programador precisa empregar um es-

força extra para codificar um mecanismo que torne possível postergar o consumo de eventos (e.g., fila de eventos). Também é preciso que o programador codifique um mecanismo de detecção de falhas extra. Uma vez que a thread do cliente realiza a chamada remota e desbloqueia, não existe qualquer mecanismo de detecção de falhas que monitore o servidor entre o desbloqueio do cliente e o recebimento do *callback*. Consequentemente, se não houver um mecanismo extra de detecção de falhas, o cliente não tem como diferenciar se o servidor falhou ou está apenas demorando a realizar o *callback*.

Pode-se evitar os problemas de objetos distribuídos através do uso de soluções baseadas em troca de mensagens, tais como MOMs, EBMs, MPI, PVM, *sockets* ou Twisted, discutidas na Seção 2.2. Todavia, com exceção do Twisted, nenhuma das soluções oferece boa integração com a linguagem de programação. O uso direto de *sockets*, por sua vez, requer uma porta aberta no *firewall* para cada entidade remota com a qual se comunica, além de a detecção de falhas acontecer através de *timeouts* que não podem ser modificados em tempo de execução. MOMs e EBMs, apesar de poderem usar a entidade intermediária (servidor) como único ponto de comunicação (sendo necessário apenas uma porta aberta no *firewall* por domínio administrativo), abordam detecção de falhas através de abordagens extremas: deixar para aplicação, ou considerar que falhas nunca acontecem.

Porém, Twisted permite um modelo de comunicação assíncrono bem integrado à linguagem de programação Python. De fato, das diversas infra-estruturas pesquisadas durante nosso levantamento bibliográfico, Twisted é a que mais se parece com o JIC. Na verdade, descobrimos esta solução quando estávamos em um estado bem avançado de nossa pesquisa. Trata-se de uma abordagem que, assim como o JIC, consegue combinar objetos distribuídos com uma arquitetura *event-driven*, eliminando os problemas com threads discutidos neste capítulo. Portanto, enxergamos Twisted como um reforço a nossa tese.

Entretanto, JIC e Twisted possuem algumas diferenças. Primeiramente, temos que Twisted é um *framework* e JIC é uma solução. Por isso, Twisted permite vários estilos de comunicação e pode ser adaptado aos ambientes os quais não contempla. JIC foi desenvolvido para contemplar um tipo específico de aplicações (aplicações distribuídas não-bloqueantes), não sendo tão flexível quanto o Twisted. No que diz respeito à detecção de falhas, o JIC oferece um mecanismo flexível e configurável, baseado no *framework* descrito em [FDGO99]. Atualmente, o JIC implementa um mecanismo de detecção de falhas que se adapta a modi-

ficações no ambiente e que usa um modelo probabilístico [HDYK04] para tomar decisões a respeito do estado de uma entidade. O Twisted, por sua vez, possui um mecanismo de detecção de falhas dependente do protocolo que está sendo usado. Se a comunicação é feita via TCP, por exemplo, Twisted detecta uma falha após o "estouro" do *timeout* usado por TCP. Esta detecção é repassada para a aplicações através do envio de exceções para objetos `Deferred`s, como mostrado na Seção 2.3.3.

Uma outra diferença diz respeito ao protocolo usado. Twisted é flexível, permitindo a escolha do protocolo de comunicação a ser usado. JIC, por sua vez, permite apenas invocações remotas assíncronas, sendo mais específico.

Em relação à comunicação na presença de *firewalls* e NATs, JIC possui naturalmente uma entidade intermediária (um servidor, ou *relay* de comunicação). Esta entidade funciona como ponto de comunicação para um domínio, de forma que apenas este servidor precisa estar acessível através do *firewall*, i.e., não é necessário abrir mais de uma porta no *firewall* independente da quantidade de aplicações internas ao *firewall* que se comunicam com o mundo externo. Twisted, apesar de não possuir um servidor inerente ao seu estilo de comunicação, pode construir um protocolo especializado que faça uso de entidades intermediárias para a comunicação, assim como o JIC faz.

No que diz respeito à comunicação assíncrona, o JIC requer que o programador construa explicitamente a interação entre entidades, definindo métodos de *callback* para cada invocação remota assíncrona e métodos explícitos para notificação de falhas (como veremos na Seção 4.5.6). Twisted faz o uso de `Deferred`s para permitir a gerência de estados do componente, seja via métodos de *callback*, seja via métodos de *errorback*. Não sabemos ainda se uma das duas opções possibilita melhor disciplina ao programador, ou se a escolha de uma ou outra se adequa a determinado tipo de aplicação. Acreditamos que esta investigação pode ser feita em trabalhos futuros, de forma que a experiência com estas duas infra-estruturas nos ajudará a entender suas vantagens e desvantagens quando comparadas uma à outra.

Por fim, observamos que o JIC aparece como uma solução capaz de endereçar imediatamente todos os problemas que enfrentamos durante o desenvolvimento do OurGrid, discutidos na Seção 1.1. Twisted também é capaz de endereçar tais problemas, mas precisa de algumas adaptações, como por exemplo, a implementação de um mecanismo de detecção de falhas configurável e um protocolo de comunicação que permita comunicação na presença

de *firewalls* e NATs. Uma outra restrição está no fato de que o OurGrid é implementado em Java e Twisted foi projetado para ser bem integrado à linguagem Python.

Capítulo 4

Solução proposta: *Java Internet Communication*

Neste capítulo, apresentamos o JIC (*Java Internet Communication*) [LCBF06], solução que endereça os problemas discutidos no Capítulo 3. O principal objetivo do JIC consiste em oferecer acesso não-bloqueante a objetos Java localizados em um sistema distribuído, de maneira que threads tenham um escopo bem definido, além de preservar boa integração com a linguagem de programação, permitindo que o mecanismo de checagem de tipos funcione entre objetos que residem em processos diferentes.

Para que a solução funcione corretamente em um sistema distribuído, é preciso usar um mecanismo de detecção de falhas que possa monitorar as entidades remotas. Note que não é possível embutir a detecção na invocação, tal qual objetos distribuídos fazem, pois as invocações através do JIC não são bloqueantes. A solução proposta possui um mecanismo de detecção de falhas sofisticado, que é fácil de usar e possui semântica bem definida.

Além disso, diante das características de sistemas distribuídos, os quais frequentemente ultrapassam vários domínios administrativos, é desejável que a solução proposta simplifique a comunicação nos ambientes em que há presença de *firewalls* e NATs. Consideramos que tornar possível a comunicação em tais ambientes significa oferecer um mecanismo que atravesse *firewalls* e NATs (e.g., tunelamento HTTP [And02] ou uso de *relay*), ou seja requerido um esforço administrativo mínimo para implantar a solução entre múltiplos domínios administrativos (e.g., não exigir mais do que uma única porta aberta no *firewall*). Como veremos adiante, a solução proposta usa uma entidade intermediária para realizar comunicação, evi-

tando que seja necessário abrir mais do que uma porta no *firewall*.

4.1 Arquitetura

A arquitetura do JIC é formada por dois componentes principais: ***Event Processor*** e ***Access Point***. Um *Event Processor* representa um objeto Java que pode ser acessado remotamente de forma assíncrona. Um *Access Point* agrupa um conjunto de *Event Processors* e delimita escopo para as threads que os acessam, conforme veremos a seguir.

4.1.1 Access Point

O *Access Point* é o ponto de entrada para um conjunto de objetos JIC. Isso significa que, para receber invocações remotas, um *Event Processor* precisa estar associado a um *Access Point*. Ao associar um *Event Processor* a um *Access Point*, dizemos que o *Event Processor* está *exportado*, ou seja, está remotamente acessível através deste *Access Point*.

Cada *Access Point* possui um *objeto de comunicação*, uma *fila de invocações*, um *conjunto de threads* e um módulo de detecção de falhas, assim como apresentado na Figura 4.1. O objeto de comunicação é responsável por enviar e receber mensagens em favor dos *Event Processors* locais. Conforme veremos adiante, na implementação atual, o objeto de comunicação usa Jabber [JSF06] como infra-estrutura para troca de mensagens. Entretanto, quaisquer outras infra-estruturas de comunicação que permitam troca de mensagens de maneira assíncrona (e.g., EBMs e MOMs) poderiam ser usadas por este objeto. Ao receber uma mensagem, o objeto de comunicação converte tal mensagem em um *evento JIC* e o coloca na fila de invocações. Cada evento colocado na fila representa uma invocação a ser realizada em algum *Event Processor* que está exportado no *Access Point*. O conjunto de threads do *Access Point*, chamadas *Event Handlers*, é responsável por consumir e processar os eventos da fila de invocação. Tais threads permanecem em *loop* infinito, de forma que removem um evento da fila, interpretam o método que o evento representa e executam a invocação no *Event Processor* alvo. O processamento dos eventos da fila obedece ordem FIFO (*First In, first Out*, ou seja, primeiro a entrar, primeiro a sair).

Logo, o processamento de uma invocação remota acontece da seguinte maneira: seja um método m, declarado na interface do Objeto JIC obj, que está exportado no *Access Point* AP.

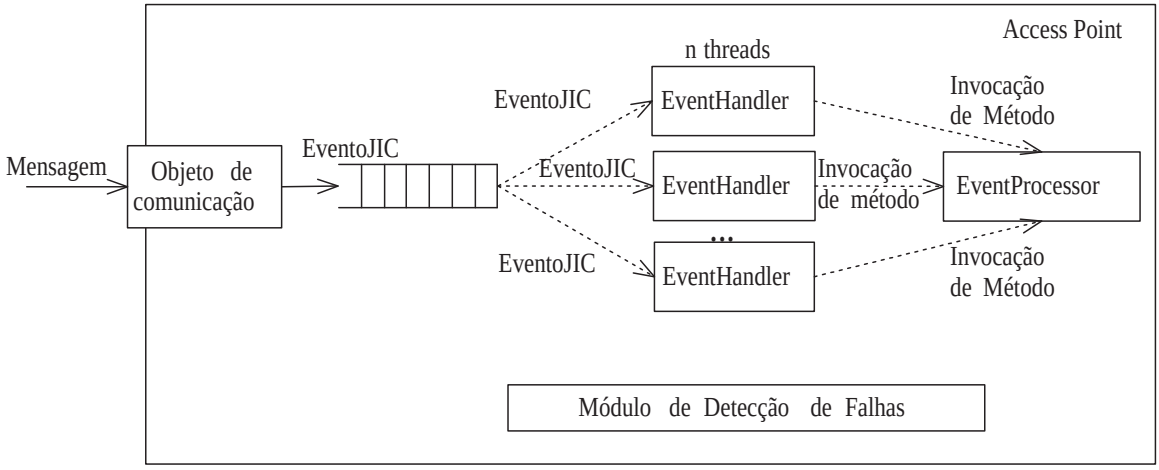


Figura 4.1: Arquitetura de um *Access Point*

Ao invocar este método remotamente, a infra-estrutura JIC do lado cliente encapsula a invocação (parâmetros reais, formais, etc) em um evento JIC e usa o objeto de comunicação para enviar este evento ao *Access Point* AP. O evento, ao ser recebido por AP, será armazenado em sua fila de invocações. Uma das threads de AP irá remover o evento da fila, interpretar o método que o evento representa e o *Event Processor* alvo, desencapsular parâmetros reais e finalmente invocar o método m em obj. É importante salientar que, caso AP possua t threads, e existirem $t + u$ invocações que ainda não foram processadas, então cada uma das t threads está consumindo um evento, de forma que os últimos u eventos permanecem na fila, esperando que alguma thread termine o processamento de um evento e volte a consumir da fila de invocações. Em particular, se AP possuir apenas uma thread ($t = 1$), os *Event Processors* exportados em AP não precisam implementar código multi-threaded, ou seja, o programador não precisa se preocupar com concorrência.

Note que um *Access Point* é um componente auto-contido que delimita as fronteiras para as suas threads. Logo, através do JIC, as threads não executam por toda a aplicação distribuída. Ao invés disso, cada thread permanece confinada dentro do *Access Point* ao qual pertence. É exatamente esta característica que faz com que threads sejam mais fáceis de tratar em aplicações construídas sobre JIC. Dessa forma, aplicações que usam JIC são componentizáveis, i.e., o programador não precisa raciocinar sobre o comportamento das threads por toda a aplicação distribuída. É necessário raciocinar apenas sobre um *Access Point* por vez. No entanto, se dois *Access Points* estão em um mesmo processo (mesma máquina virtual

java), o programador precisa garantir que threads de um *Access Point* não acessam diretamente objetos que pertencem a outro *Access Point*. Objetos de *Access Points* diferentes, até nas situações em que estão em um mesmo processo, devem se comunicar apenas através do JIC, ou seja, através de eventos que são armazenados na fila de invocações. Um trabalho em andamento no sentido de garantir tal isolamento consiste em aplicar maneiras automáticas de instrumentar o código e usar técnicas de monitoração para detectar possíveis violações desta necessidade [DDHM02]. Além disso, é possível explorar outras abordagens que visam promover isolamento entre aplicações que executam em um mesmo processo, através de modificações na máquina virtual [Krz02].

É importante notar que, *Access Points* possuem identificações globalmente únicas, o que diferencia quaisquer dois *Access Points* em qualquer parte do sistema distribuído. Como veremos adiante, o esquema de identificação usado pelos *Access Points* aproveita a infraestrutura de troca de mensagens subjacente (no caso, Jabber).

4.1.2 Serviços e Objetos

Conforme explicado anteriormente, um *Event Processor* representa um *objeto* que pode ser acessado remotamente via JIC. Tais objetos oferecem *serviços* remotos, ou seja, funcionalidades que podem ser invocadas por outros programas. Intuitivamente, serviços e objetos podem ser representados por um único componente arquitetural, ou seja, o objeto (*Event Processor*) que oferece a funcionalidade é o próprio serviço. No entanto, precisamos diferenciar estes dois componentes no JIC, pois um objeto pode sair do ar e ser substituído por outro, que provê o mesmo serviço que era oferecido anteriormente. De fato, objetos saem do ar frequentemente por várias razões: queda de energia, instalação de correções, rejuvenescimento de software, etc.

Portanto, conceitualmente, diferenciamos serviços e objetos da seguinte maneira: um serviço é uma entidade permanente que realiza um determinado papel no sistema distribuído (e.g., um serviço de nomes de um domínio). Um objeto, por sua vez, é o *objeto Java* que em determinado momento está encarregado de prover o serviço. Na prática, serviços e objetos são diferenciados através de suas identificações. A identificação de um serviço é formada pela identificação do *Access Point* ao qual o serviço pertence, mais o nome do serviço, que é específico da aplicação. Uma vez que a identificação do *Acess Point* é globalmente única

e o JIC não permite dois serviços com o mesmo nome em um mesmo *Access Point*, a identificação de um serviço também é globalmente única. A identificação de um objeto também é globalmente única e é formada pela identificação do serviço junto com um *número de encarnação*. O número de encarnação representa a instância do objeto Java que executa o serviço em um dado momento.

O uso de números de encarnação é bastante útil para diferenciar o reinício de um serviço da indisponibilidade temporária do mesmo. Por exemplo, existem situações em que o objeto que implementa um serviço é capaz de persistir seu estado. Sendo assim, caso tal objeto saia do ar, o objeto que o substitui é capaz de recuperar o último estado armazenado, podendo assim prosseguir com o trabalho iniciado pelo objeto anterior. Para que isso seja possível, o JIC permite que os objetos escolham o seu número de encarnação na hora de serem exportados. Portanto, um objeto pode escolher o mesmo número de encarnação usado pelo objeto anterior, fazendo com que a troca de objetos pareça apenas uma indisponibilidade temporária do serviço. Porém, nas situações em que o objeto que implementa o serviço sai do ar e não armazena estado, não é possível que o objeto substituto recupere o estado anterior. Este novo objeto, por conseguinte, não possui qualquer histórico das interações passadas com outros objetos remotos. Isso significa que tais objetos remotos precisam saber que o objeto com o qual estavam interagindo não existe mais. Neste caso, quando o serviço reinicia, o novo objeto deve optar por receber um novo número de encarnação gerado pelo JIC no momento da exportação, indicando que o novo objeto que implementa o serviço não possui informação sobre as interações passadas com este serviço. De fato, se o serviço não armazena estado, os objetos que o usam precisam tomar decisões, como por exemplo, submeter novamente as requisições pendentes.

Note que a comunicação no JIC acontece sempre entre objetos, ou seja, não existe serviço ativo sem que haja um objeto exportado que o represente. Sendo assim, através da identificação de um serviço, o JIC permite obter a referência para o objeto que o está executando em um dado momento.

4.1.3 Detecção de Falhas

Um outro componente que aparece na arquitetura é o detector de falhas. Conforme afirmado anteriormente, uma vez que não é possível usar os *timeouts* de comunicação da maneira que

as soluções bloqueantes usam, é preciso projetar um mecanismo de detecção de falhas mais sofisticado. Na verdade, apenas usar os *timeouts* da camada de comunicação não é uma solução sofisticada, pois não se tem flexibilidade para lidar com qualidade de serviço, como por exemplo, em relação ao tempo que se gasta para detectar uma falha, ou taxa de falsas suspeições realizadas pelo mecanismo de detecção de falhas em um dado ambiente.

Portanto, o JIC implementa um detector de falhas distribuído capaz de balancear o *trade-off* entre tempo de detecção e taxa de falsas suspeições. Tal mecanismo é flexível e configurável, baseado no modelo probabilístico descrito em [HDYK04] e explicado na Seção 2.4.1. Consideramos que o mecanismo é flexível porque foi construído como um *framework*, que pode ser facilmente estendido para implementar outras estratégias de detecção. Além disso, é configurável porque oferece ao usuário a possibilidade de definir valores para os parâmetros que influenciam a qualidade de serviço da detecção. Como veremos na Seção 4.5.6, precisamos entender melhor o efeito do mecanismo atual implementado no JIC. Logo, estamos usando um mecanismo mais simples por enquanto, no qual é possível configurar dois parâmetros: *intervalo entre heartbeats* (Δ_i) e *timeout de detecção* (Δ_{to}). Estes parâmetros podem ser incluídos pelo usuário em um arquivo de configuração. Através deles, é possível balancear tempo de detecção e taxa de falsas suspeições.

Como podemos ver através da Figura 4.1, cada *Access Point* possui um módulo do serviço de detecção de falhas distribuído. Cada módulo é responsável por fornecer informações sobre os *Event Processors* que pertencem ao seu *Access Point*. Logo, nota-se que o monitoramento de objetos e serviços é realizado na prática entre *Access Points* e não entre objetos. Objetos são apenas notificados sobre a mudança de estado de outros objetos, desde que haja interesse. É importante ressaltar que um objeto JIC pode se interessar na falha de um outro objeto ou no "aparecimento" (que inclui recuperação ou reinício) de um serviço. O *Access Point* fornece uma interface de alto nível que permite o registro de tais interesses, ou o cancelamento de um registro previamente definido.

Uma vez que um objeto A exportado através do *Access Point* AP_A registra interesse na falha do objeto B, exportado através do *Access Point* AP_B, o mecanismo de detecção de falhas do JIC começa a monitorar o objeto B. Este monitoramento é realizado através de um modelo *pull*, em que o módulo detector de falhas do *Access Point* AP_A periodicamente solicita informações ao módulo detector de falhas de AP_B sobre o estado do objeto B. O

módulo de detecção em AP_B recebe as solicitações e responde se B está exportado ou não. Respostas negativas implicam suspeições. Ainda, se o módulo do detector em AP_A não recebe a resposta dentro de um *timeout* Δ_{to} definido de acordo com a qualidade de serviço da aplicação, considera que o *Access Point* AP_B está fora do ar, pois não foi capaz de responder. Dessa forma, AP_A suspeita de que B falhou, notificando A sobre este fato. Situação análoga acontece quando um objeto A se interessa na recuperação de um serviço B. Porém, a notificação acontece somente quando o *Access Point* AP_B responde a AP_A que existe um objeto exportado oferecendo o serviço B.

Uma vez que um objeto é notificado sobre a falha de um outro objeto, ou recuperação de um serviço, o interesse é automaticamente descadastrado, ou seja, a notificação acontece uma única vez por objeto.

4.2 Conectividade

Como mencionado na Seção 4.1.1, o JIC usa Jabber como infra-estrutura para troca de mensagens, ou seja, todos os *Event Processors* de uma aplicação JIC se comunicam através de Jabber. O principal motivo que nos guiou a esta escolha está no fato de que Jabber se apresenta como uma ótima solução para troca de mensagens na presença de *firewalls* e NATs. Além disso, trata-se de uma solução que evoluiu em direção à padronização. Os protocolos que compõem o Jabber foram formalizados pela IEFT [IET06a] sob o nome de XMPP (*eXtensible Messaging and Presence Protocol*) [XMP06].

Jabber é um conjunto de protocolos baseados em XML, principalmente usados para troca de mensagens instantâneas, permitindo que entidades troquem mensagens e informações sobre presença de maneira próxima ao tempo real. Sua arquitetura é semelhante à arquitetura de e-mail, como mostrado na Figura 4.2.

Duas entidades que pretendem se comunicar via Jabber devem criar uma conta em um servidor Jabber, de forma que cada uma recebe uma identificação, conhecida como JID (*Jabber IDentification*). Uma entidade que possui um JID e se conecta a um servidor Jabber é chamada *cliente Jabber*. No JIC, cada *Acess Point* é um cliente Jabber. JIDs são baseados em DNS (*Domain Name System*) e em esquemas URI (*Universal Resource Identifier*) reconhecidos. Sendo assim, além de identificarem globalmente uma entidade, JIDs são facilmente

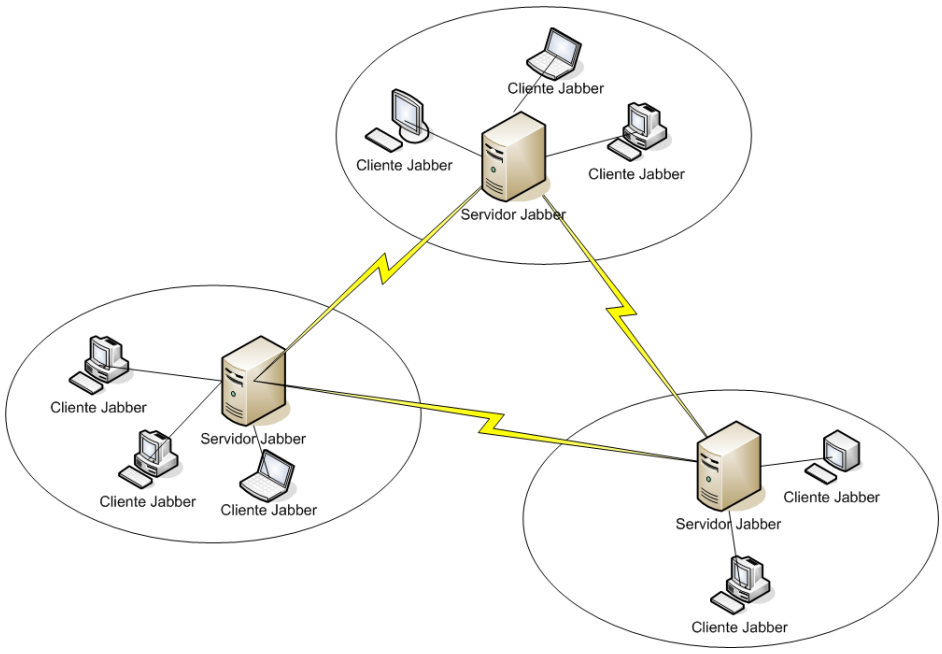


Figura 4.2: Arquitetura Jabber.

compreensíveis por seres humanos e têm a forma parecida com a de um endereço de e-mail, como mostrado abaixo:

$$usuario@dominio/recurso$$

No exemplo acima, $dominio$ representa um FQDN (*Full Qualified Domain Name*) para o servidor Jabber, $usuario$ representa o cliente Jabber que se conecta ao servidor e $recurso$ representa um objeto específico da sessão que pertence a um cliente. Então, para enviar uma mensagem para um cliente C_2, um cliente Jabber C_1 a envia para o seu servidor, que a repassa para o servidor ao qual C_2 está conectado, que finalmente entrega tal mensagem para C_2, como mostra a Figura 4.3.

Note que o uso do servidor Jabber como *relay* diminui bastante os problemas impostos pelo uso de *firewalls* e NATs. Basicamente, tais problemas se devem ao fato de que estas abordagens limitam a conectividade entre pares que se comunicam, bloqueando conexões ou fazendo com que o início de uma conexão seja possível apenas a partir de uma direção. Em geral, caso a porta usada por um objeto remoto esteja fechada no *firewall* de seu domínio administrativo, uma entidade fora deste domínio não pode iniciar uma conexão com este objeto remoto. Uma vez que todo *Event Processor* se comunica através de um servidor Jabber, é preciso abrir apenas uma porta por domínio administrativo, não importa quantos

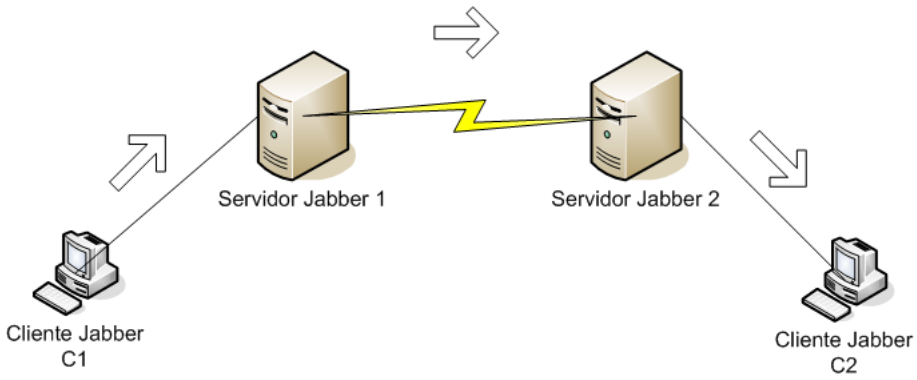


Figura 4.3: Fluxo de comunicação entre o cliente Jabber C_1 e o cliente Jabber C_2

objetos são exportados pela aplicação. Esta porta é exatamente a porta que torna o servidor Jabber acessível por outros servidores Jabber. Ainda, se a porta para mensagens instantâneas (5222) de um domínio já estiver aberta a priori, a comunicação através de Jabber para este domínio pode ser feita sem nenhum esforço administrativo extra.

Através do uso do Jabber, também é possível usar comunicação segura através de TLS (*Transport Layer Security*) [IET06e] e SASL (*Simple Authentication and Security Layer*) [IET06c].

4.3 Semântica

Entender *precisamente* a semântica que a infra-estrutura de comunicação oferece é indispensável para se desenvolver aplicações robustas. Desenvolver aplicações com o JIC não foge a esta regra. É preciso entender a semântica fornecida pelo JIC para que seja possível compreender como acontece a comunicação entre objetos e em que situações a infra-estrutura de comunicação detecta falhas.

Considere dois objetos remotos A e B, os quais estão exportados nos *Access Points* AP_A e AP_B, respectivamente. Tais objetos pretendem se comunicar um com o outro. Os *Access Points* AP_A e AP_B podem ser o mesmo, podem ser *Access Points* distintos no mesmo espaço de endereçamento, ou ainda em espaços de endereçamento diferentes, possivelmente em máquinas distintas. Cada *Access Point* mantém uma conexão ativa com seu respectivo servidor Jabber. Os objetos A e B se comunicam por invocações remotas de métodos, de forma que uma invocação é "empacotada" em uma mensagem (através de um mecanismo

de *serialização*) e enviada para o lado receptor, onde é armazenada na fila e consumida em seguida, por ordem de chegada. Todos os parâmetros da invocação são passados por valor, exceto *Event Processors*, que são passados por referência. Dado este cenário, temos as seguintes definições:

O *Serviço de Comunicação Jabber* usado por dois objetos JIC quaisquer corresponde ao conjunto de servidores Jabber e canais de comunicação (*links*) que participam da comunicação entre tais objetos. Cada canal de comunicação e servidor Jabber é chamado *elemento do Serviço de Comunicação Jabber*. Os canais de comunicação usados entre um cliente Jabber e o servidor Jabber usam TCP para a troca de mensagem. O mesmo acontece com os canais de comunicação entre servidores.

Dizemos que um canal de comunicação do serviço Jabber está *ativo* quando este é capaz de entregar mensagens entre as duas entidades que interliga (e.g., cliente Jabber - servidor Jabber). Caso o canal não seja capaz de entregar tais mensagens, dizemos que está *inativo*.

Sejam S_A e S_B os servidores Jabber usados pelos *Access Points* A e B, respectivamente, um *caminho* entre dois objetos JIC A e B é composto por todos os elementos do *Serviço de Comunicação Jabber* envolvidos na comunicação entre eles. Logo, podemos dizer que **existe caminho** entre A e B se e somente se S_A e S_B estão no ar e todos os canais de comunicação do Serviço de Comunicação Jabber entre eles estão ativos. Isso significa que, se existe caminho entre A e B, eles podem se comunicar, caso estejam devidamente exportados nos *Access Points* AP_A e AP_B. De maneira análoga, dizemos que **não existe caminho** entre A e B se S_A ou S_B estiver fora do ar ou se existir pelo menos um canal de comunicação pertencente ao Serviço de Comunicação Jabber inativo. Um caminho contém no mínimo um servidor Jabber (quando os objetos estão conectados a um mesmo servidor) e no máximo dois servidores Jabber (quando os objetos estão conectados a servidores distintos). Considere os servidores Jabber, canais de comunicação e objetos JIC apresentados na Figura 4.4:

Caso exista caminho entre A e B, dizemos que A está conectado a B, o que podemos denotar por:

$$conectado_B(A) = funcionando(S_A) \ \wedge \ funcionando(S_B) \ \wedge \ ativo(C_{A-S_A}) \ \wedge \ ativo(C_{S_A-S_B}) \ \wedge \ ativo(C_{S_B-B})$$

Obviamente, o conceito de caminho não tem qualquer valor caso os objetos não este-

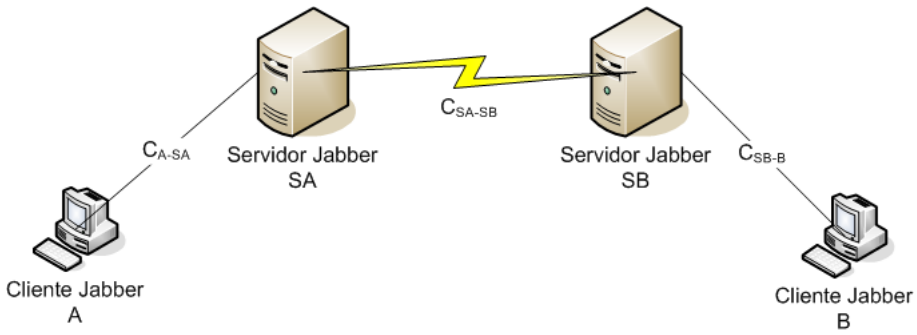


Figura 4.4: Cenário de Comunicação entre objetos A e B

jam *disponíveis*. Portanto, dizemos que um objeto JIC está **disponível** se este objeto está exportado através de um *Access Point*. Da mesma forma, dizemos que um objeto JIC está **indisponível** se este objeto não está exportado através de qualquer *Access Point*. Note que, se o *Access Point* através do qual o objeto foi exportado está fora do ar, o objeto está *indisponível*. Para um objeto A, a notação do seu estado é representada por $disponivel(A)$.

Diante das definições acima, podemos definir um conceito relacionado a *alcance* entre objetos JIC, que será de grande importância para o entendimento da semântica do JIC:

$$alcancavel_A(B) = disponivel(A) \ \wedge \ disponivel(B) \ \wedge \ conectado_A(B),$$

ou seja, B é **alcançável** por A se e somente se, A está disponível, B está disponível e B está conectado a A.

Dadas estas definições, o JIC provê uma semântica precisa para os programadores que escrevem aplicações distribuídas. Considerando que o *Event Processor* A registrou interesse na falha do *Event Processor* B e vice-versa, o JIC garante que:

1. Se B permanece alcançável por A, então **toda mensagem enviada de *A* para *B* é recebida em algum momento no futuro por** B;

2. Se B torna-se e permanece inalcançável a partir de A, então A será notificado em algum momento no futuro de que B está *inalcançável*. Neste caso, dizemos que ***A* suspeita de *B***.

3. Se existe a perda de pelo menos uma mensagem enviada de A para B, A será notificado em algum momento no futuro de que B está inalcançável. Além disso, B também será

notificado em algum momento no futuro de que A está inalcançável, i.e., ***A* suspeita de *B*** e ***B* suspeita de *A***.

Note que "A suspeita de B" não nos dá garantia de que B realmente está *inalcançável*. Existe a possibilidade de que uma mensagem de A para B tenha sido descartada pela rede, ou que B esteja apenas respondendo às mensagens de detecção de falha de maneira muito lenta. Isso significa que o programador, ao raciocinar sobre a aplicação, deve ter em mente que (i) ou as chamadas remotas são invocadas com sucesso, ou (ii) o objeto que disparou a invocação será notificado de uma suspeita. Todavia, ser notificado da suspeita de algum objeto não garante que a falha realmente aconteceu.

Diante disso, é preciso garantir que a semântica acima se mantém sempre verdadeira, assegurando que não há comportamentos inconsistentes no decorrer da execução de aplicações que usam JIC. Apresentamos, portanto, os mecanismos usados para garantir que a semântica é sempre verdadeira e como tais mecanismos garantem tal semântica.

A semântica definida através dos itens 1 e 2 é facilmente garantida. O item 1 pode ser garantido pela própria camada de comunicação. Já o item 2 é assegurado através do mecanismo de detecção de falhas apresentado na Seção 4.1.3.

Para o item 1, uma vez que ambos os objetos estão *disponíveis* e *existe caminho* entre eles, uma mensagem enviada de A para B é sempre entregue em algum momento no futuro, devido à garantia de entrega da conexão TCP entre cada entidade que constitui o fluxo de mensagens.

Para o item 2, caso um dos objetos não esteja alcançável por não estar mais exportado, o detector de falhas que está monitorando o objeto irá detectar a falha, pois receberá uma resposta de que o objeto de interesse não está exportado. Caso um dos objetos não esteja alcançável porque não está mais conectado ao objeto interessado, o detector de falhas deste último objeto também irá detectar a falha, pois não receberá respostas às suas mensagens de monitoramento dentro de um tempo limite previamente estabelecido, notificando assim o objeto interessado.

Contudo, existe uma situação em que B permanece *alcançável* a partir de A, mas deixa de receber uma mensagem devido a uma breve falha na rede (ou de um servidor Jabber). Neste caso, o detector de falhas do *Access Point* AP_B pode ter sido capaz de responder todas as mensagens de monitoramento oriundas de AP_A, mesmo que B tenha falhado em receber

uma mensagem de aplicação. De fato, AP_A precisa detectar esta falha, para notificar o objeto A de que B não está alcançável, de acordo com o item 3 da definição da semântica. Porém, o detector de falhas, da maneira que foi projetado, não consegue detectar tal falha. Por causa disso, implementamos numeração de mensagens e um protocolo de conexão em três vias (*three-way handshake protocol*) semelhante ao protocolo de conexão do TCP [Tom75]. Através deste protocolo, quaisquer dois *Access Points* precisam estabelecer conexão antes de trocarem a primeira mensagem. Basicamente, o protocolo define números de sequência para as mensagens de aplicação entre dois *Access Points*. Logo, é possível identificar perda de mensagens de aplicação. Uma vez que as mensagens são entregues obedecendo à ordem de envio, um número de sequência maior do que o esperado caracteriza a perda de uma mensagem. Nesta situação, o *Access Point* que identificou a perda de uma mensagem (AP_B), libera a conexão (também usando um protocolo de três vias) e notifica B de que A está inalcançável. O mesmo acontece com AP_A ao terminar de executar o protocolo que libera a conexão (iniciado por AP_B). Portanto, temos que ambos A e B são notificados de que o outro não está alcançável, assim como definido no item 3 da semântica. Vale salientar de que a aplicação não precisa saber que mensagem específica foi perdida. Precisa apenas ser notificada de que o objeto de interesse falhou.

Note que, apesar de estarmos usando TCP entre *Access Points* e os servidores Jabber, não podemos simplesmente usar o protocolo implementado sobre TCP para identificar perdas de mensagens. Isso acontece porque não temos uma conexão TCP fim-a-fim entre *Access Points*. Ao contrário, temos uma conexão TCP ponto-a-ponto entre cada *Access Point* e seu respectivo servidor e entre os próprios servidores.

4.4 Modelo de Programação

Uma vez que o JIC é baseado em uma arquitetura *event-driven*, o programador é levado a construir cada componente de sua aplicação como uma máquina de estados. Sendo assim, cada invocação remota exportada por um objeto é vista como um evento. Para cada evento, o programador deve decidir como tratá-lo, considerando o estado atual de sua aplicação. Além disso, o JIC define eventos próprios, de notificação de falhas, i.e., eventos inerentes a sistemas distribuídos que precisam ser tratados pela aplicação. Logo, o programador também

é levado a raciocinar sobre o tratamento de eventos de falha. Acreditamos que esta é uma boa prática de programação, fazendo com que o programador tenha mais disciplina e consiga evitar problemas que antes só eram detectados após algumas rodadas de teste.

Um dos objetivos do JIC é manter boa integração com a linguagem de programação. Definimos anteriormente que atingir boa integração com a linguagem significa permitir que o mecanismo de checagem de tipos funcione para objetos remotos. A presença de um mecanismo de checagem de tipos contribui bastante para a captura de *bugs* em fases incipientes do desenvolvimento de software. Além disso, através deste mecanismo, é possível que o desenvolvimento de aplicações distribuídas tenha características mais próximas do desenvolvimento de aplicações locais. De fato, um objetivo mais geral levado em consideração ao se desenvolver o JIC consiste em manter para o modelo de objetos distribuídos o máximo possível da semântica usada pelo modelo de objetos locais. As diferenças entre os dois modelos devem ser inseridas apenas onde necessário (i.e., onde realmente existem) e devem ser explícitas, de forma que o programador tenha consciência de tais diferenças. Assim, estamos minimizando a complexidade de se implementar soluções distribuídas, sem esconder suas características.

Nossa abordagem segue então os mesmos objetivos do modelo de objetos distribuídos. Porém, o JIC lida com invocações remotas não-bloqueantes, diferentemente do modelo em comparação. Por isso, o JIC impõe algumas restrições, como a exigência de que métodos remotos sejam *void* (i.e., sem retorno) e precisa lidar com alguns aspectos de maneira diferente (e.g., detecção de falhas). Aqui, apresentamos o modelo de programação usado por um desenvolvedor que constrói aplicações usando JIC.

4.4.1 Interfaces Remotas

No modelo oferecido pelo JIC, o desenvolvedor deve programar para interfaces, não para implementações. Isso significa que clientes de objetos remotos interagem com interfaces remotas, não com classes que podem ser acessadas remotamente. Diante disso, todo objeto JIC deve prover uma interface remota, a qual estende a interface `EventProcessor`. Esta última interface apenas indica que o objeto que a implementa pode ser exportado via JIC (Assim como a interface `Remote` indica um objeto remoto em Java RMI). Esta interface contém quatro métodos que devem ser implementados por todo objeto JIC: `setObjectID`,

`getObjectID`, `setAccessPoint` e `getAccessPoint`. Estes são os métodos necessários para que o JIC gerencie um objeto remoto. Considere o fragmento abaixo do *Event Processor* que realiza a submissão de aplicações (i.e., *jobs*) no código do OurGrid:

```
public interface MyGrid extends EventProcessor{
    public void addJob(JobSpecification jobSpec);
    ...
}
```

O trecho de código acima representa a definição de uma interface remota chamada `MyGrid`. Esta interface possui um método remoto (`addJob(JobSpecification)`). Note que este método, além de não possuir retorno, também não lança exceções. Esta é outra exigência do JIC.

Estas duas exigências se devem à natureza não-bloqueante das invocações realizadas pelo JIC. Através de métodos *void*, o cliente pode prosseguir imediatamente para a próxima linha de código, sem esperar por um retorno. Se lado remoto precisar enviar algum retorno referente a uma chamada remota, deve fazer isso via *callback*. Uma vez que o cliente retorna rapidamente (antes de a invocação ser realmente processada pelo lado remoto), não faz sentido definir exceções que capturem falhas de comunicação, assim como a solução de objetos distribuídos faz.

Note que estas não são as únicas diferenças entre objetos JIC e objetos Java. Uma diferença mais sutil e mais importante está no fato de que a thread do cliente não atravessa as suas fronteiras, ou seja, não continua no lado servidor. Uma invocação apenas marca o método invocado para ser executado pela própria thread do servidor (conforme mostrado na Seção 4.1.1). Note também que, além de permitir checagem de tipos, o JIC também oferece "serialização" automática dos objetos que são passados como parâmetro no método, bem como do próprio método. Isso evita que o programador codifique as "tediosas" classes de mensagens para encapsular uma chamada de método. É importante salientar que a serialização automática e o mecanismo de checagem de tipos são possíveis devido ao uso de *stubs*, que são representações locais para objetos remotos. Conforme veremos adiante, *stubs* são gerados automaticamente em tempo de execução pelo JIC.

4.4.2 Implementação de Interfaces Remotas

Existem duas maneiras de implementar uma interface remota. A primeira delas, consiste em reusar a implementação remota definida a priori pelo JIC, conforme ilustrado através da Figura 4.5. A implementação "padrão" oferecida pelo JIC é denominada `SimpleEventProcessor`. Esta classe possui uma implementação padrão para os métodos presentes na interface `EventProcessor`, além de fornecer semântica especializada e apropriada para objetos remotos pois sobrescreve a implementação dos métodos `equals`, `toString` e `hashCode` de objetos Java. A classe `SimpleEventProcessor` foi desenvolvida para evitar que o programador tenha o trabalho de implementar os métodos mencionados acima para cada *Event Processor* da aplicação.

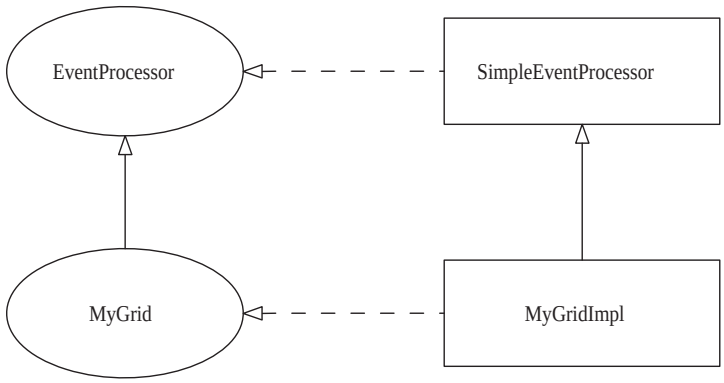


Figura 4.5: Reuso da implementação remota oferecida pelo JIC

Alternativamente, o programador pode não querer reusar a implementação remota fornecida pelo JIC. Neste caso, o código do objeto remoto não estende a classe `SimpleEventProcessor`, como mostrado na Figura 4.6. A vantagem obtida por este esquema está no fato de que, como Java não oferece herança múltipla, o objeto remoto tem a "liberdade" de estender qualquer outra classe que não seja `SimpleEventProcessor`. O lado negativo, contudo, está no fato de que o objeto remoto deverá implementar diretamente os métodos da interface `EventProcessor`, bem como ser responsável por fornecer sua semântica própria para objetos distribuídos, sobrescrevendo os métodos `equals`, `toString` e `hashCode`. Recomendamos o *reuso da implementação remota*, uma vez que o programador precisa lidar menos com código transversal à funcionalidade e pode aproveitar um código maduro e bem testado oferecido pela infra-estrutura de comunicação.

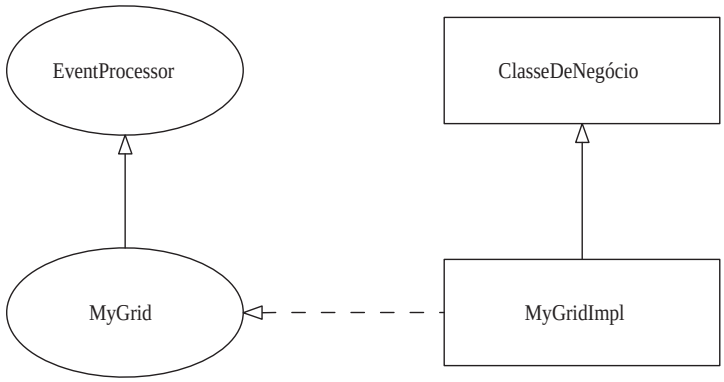


Figura 4.6: Implementação remota sem reuso da classe oferecida pelo JIC

No trecho de código abaixo, temos a implementação da interface `Mygrid`, na qual a classe `MyGridImpl` estende a classe `SimpleEventProcessor`.

```
public class MyGridImpl extends SimpleEventProcessor
                                    implements MyGrid{
    public void addJob(JobSpecification jobSpec){
       // lógica de negócio:
       ...
    }
    ...
}
```

Note que uma classe pode implementar tantas interfaces remotas quanto desejado. Além disso, uma classe também pode estender uma outra que implementa interfaces remotas. Porém, apenas os métodos que pertencem a alguma interface que estende `EventProcessor` são considerados como remotos, ou seja, são acessíveis remotamente.

4.4.3 Exportando um Objeto JIC

Após implementar um objeto JIC, apenas instanciá-lo não o faz remotamente acessível. É preciso que este objeto esteja associado a um *Access Point*, ou seja, é preciso que esteja exportado. O JIC oferece uma interface de alto nível para criação de um *Access Point* e exportação de *Event Processors*, como mostrado no trecho de código a seguir:

```
1       JICConfiguration.getInstance(arquivo.conf);
```

```
2
3       String nomeAP = "myGridAP";
4       AccessPoint accessPoint = new AccessPoint(nomeAP);
5       accesspoint.start();
6
7       MyGrid mygrid = new MyGridImpl();
8       String nomeMG = "mygrid";
9
10      accesspoint.bind(nomeMG, mygrid, 1);
```

Na primeira linha de código, recupera-se uma instância para uma configuração do JIC. Este objeto de configuração consulta um arquivo definido pelo usuário que contém informações como por exemplo, o servidor Jabber ao qual o *Access Point* irá se conectar, o *login* do usuário neste servidor e a senha usada. Um exemplo de arquivo de configuração pode ser visto na Figura 4.7. Note na linha 4 que a criação do *Access Point* usa um nome definido pela aplicação. Porém, ao definir o JID do *Access Point*, o JIC usa as informações armazenadas no objeto de configuração. Sendo assim, o *Access Point* criado na linha 4 possui o seguinte JID: `john@frodo.lsd.ufcg.edu.br/myGridAP`, conforme as informações do arquivo de configuração da Figura 4.7 e o nome do *Access Point* fornecido pela aplicação. Uma vez criado, o *Access Point* ainda não está pronto para ser usado. É preciso invocar `accesspoint.start()`, assim como mostra a linha 5. Este comando negocia a conexão com o servidor Jabber e inicia as threads (*Event Handlers*) do *Access Point*. A última linha de código exporta o objeto remoto `MyGrid` através do nome `mygrid`. Note que o terceiro parâmetro informa o número de encarnação que será usado por este objeto. Logo, o JID do objeto recém exportado será `john@frodo.lsd.ufcg.edu.br/myGridAP/mygrid/1`. Caso o terceiro argumento fosse omitido, um número de encarnação aleatório seria atribuído à identificação do objeto exportado.

4.4.4 Obtendo uma Referência Remota

A maneira mais simples de obter uma referência para um objeto JIC é através da passagem de parâmetros entre objetos remotos, conforme veremos na Seção 4.4.5. Uma ma-

```
# Nome do usuário no servidor Jabber
jic.user.name = john

# Senha para o usuário no servidor Jabber
jic.user.password = $ E6RED0

# Nome do Servidor Jabber
jic.jabber.servername = frodo.lsd.ufcg.edu.br

# Porta usada pelo servidor Jabber
jic.jabber.serverport = 5222

# Porta segura usada pelo servidor Jabber
jic.jabber.serversecureport = 5223
```

Figura 4.7: Arquivo de configuração para uma aplicação JIC: *arquivo.conf*.

neira alternativa, quando se tem a identificação do objeto remoto, consiste em usar o método `getReference`, fornecido pelo *Access Point*. Seja um *Access Point* `apCliente`, que está devidamente conectado a um servidor Jabber, é possível usar o seguinte trecho de código para obter uma referência para o objeto `MyGrid` exportado na Seção 4.4.3:

```
idDoObjeto="john@frodo.lsd.ufcg.edu.br/myGridAP/mygrid/1";
MyGrid mgRef = apCliente.getReference(idDoObjeto, MyGrid.class);
```

O método `getReference` retorna uma referência para o objeto cujo JID foi passado como primeiro parâmetro. É importante ressaltar que esta referência é obtida localmente, através da criação de um *stub* (atribuído a `mgRef`), sem realizar qualquer invocação remota. Este *stub* implementa a interface remota passada como segundo parâmetro (`MyGrid`), bem como as demais interfaces remotas que esta interface estende. Dessa forma, um *stub* não provê acesso a métodos não-remotos de um objeto. Do ponto de vista da linguagem de programação, operações de *cast* e `instanceof` para o conjunto de interfaces remotas implementadas pelo *stub* funcionam corretamente. Por exemplo, as duas linhas de código a seguir retornam ambas um valor verdadeiro:

```
mgRef instanceof MyGrid
mgRef instanceof EventProcessor
```

mas a linha de código seguinte retorna um valor falso:

```
mgRef instanceof MyGridImpl
```

Mesmo que o objeto remoto seja uma instância da classe `MyGridImpl`, o *stub* não tem ciência disso, pois o desenvolvedor programa para interfaces.

Stubs são gerados dinamicamente em tempo de execução e usam a conexão do *Access Point* local para realizar uma invocação remota. Uma vez que o cliente tem um *stub* para o objeto que implementa o serviço desejado, uma invocação é realizada de maneira simples, como mostrado abaixo:

```
mgRef.addJob(myJobSpecification);
```

As informações desta invocações são "serializadas" e enviadas ao lado remoto, de forma que a chamada será realizada no objeto `MyGridImpl`, que implementa o serviço definido na Seção 4.4.3.

Porém, existem situações em que um cliente possui a identificação para um serviço e deseja obter a referência para um objeto que implementa este serviço. Para este caso, é possível usar o mecanismo de detecção de falhas para recuperar tal referência. Conforme veremos adiante, na Seção 4.4.6, é possível usar o mecanismo de detecção de falhas do JIC para se cadastrar no aparecimento de um serviço. O mecanismo de detecção, ao detectar este aparecimento, notifica o objeto interessado, entregando uma referência (i.e., um *stub*) para o objeto que implementa o serviço, i.e., para a encarnação corrente do serviço. Portanto, através desta notificação, é possível recuperar uma referência para um objeto, quando não se sabe a encarnação do mesmo, ou seja, quando se tem apenas a identificação do serviço.

4.4.5 Passagem de Parâmetros

Assim como foi explicado na Seção 4.4.4, uma das maneiras de se obter uma referência para um objeto JIC é através da passagem de parametros em invocações remotas. O JIC garante que todo *Event Processor* passado como parâmetro em uma chamada remota é passado por referência e não por cópia (Na verdade, é passada uma cópia do *stub*). No momento da "desserialização" de uma chamada, todos os objetos que implementam direta ou indiretamente alguma interface que estende `EventProcessor` têm suas referências atualizadas com *stubs* que usam o *Access Point* local para trocar mensagens. Se um objeto passado como

parâmetro possui referência para algum *Event Processor*, esta referência também será atualizada. Note que, uma vez que *stubs* também implementam a interface `EventProcessor`, ao serem passados como parâmetro, também são atualizados com referências locais.

Porém, o mecanismo de passagem de parâmetros usado pelo JIC difere do mecanismo usado para chamadas locais no que diz respeito a objetos que não são remotos. Todo objeto local, ao ser passado como parâmetro em uma invocação remota, é passado por cópia, ou seja, no momento da "desserialização", o lado remoto possuirá sua própria cópia do objeto passado como parâmetro. Portanto, o objeto recebido pelo lado remoto não possui qualquer relação com o objeto que foi "serializado" e enviado na chamada remota. Obviamente, modificações no objeto recebido não exercem quaisquer influências sobre o objeto que foi passado como parâmetro.

4.4.6 Detecção de Falhas

Dentre as diferenças existentes entre ambientes locais e distribuídos, uma das mais importantes, e que consequentemente não pode ser escondida do programador é a presença de falhas parciais em sistemas distribuídos. Isso significa que, na hora de desenvolver uma aplicação distribuída, o programador precisa ter em mente que as entidades com as quais um componente está se comunicando podem falhar. No caso de aplicações não-bloqueantes como as construídas sobre JIC, o programador ainda precisa estar ciente de que o mecanismo de detecção de falhas não pode ser tão simples como em objetos distribuídos.

Considerando este fato, o modelo de programação do JIC oferece interfaces de alto nível, as quais permitem que o desenvolvedor da aplicação lide diretamente com as falhas de componentes distribuídos. Basicamente, um cliente JIC pode efetuar três tipos de operação: (i) registrar interesse na falha de um objeto, (ii) desregistrar interesse na falha de um objeto e (iii) tomar decisões nas situações em que é notificado sobre a falha de um objeto.

De maneira análoga, serviços cujos objetos falham podem ser exportados novamente, seja porque reiniciou, seja porque um novo objeto assumiu o papel de executar o serviço. Logo, as mesmas operações que um cliente pode realizar em relação à falha de um objeto podem ser executadas em relação à recuperação de um serviço. Um cliente JIC pode então: (i) registrar interesse na recuperação de um serviço, (ii) desregistrar interesse na recuperação de um serviço e (iii) tomar decisões nas situações em que é notificado sobre a recuperação

de um serviço.

O *Access Point* local, responsável por realizar a detecção de falhas de objetos remotos, fornece ao programador uma interface para recuperar o módulo do detector de falhas responsável pelo registro de interesses. Uma vez que um objeto manifesta interesse em relação a outro objeto ou serviço, o objeto interessado precisa ser notificado sobre a falha ou recuperação da entidade de interesse. Para que isso seja possível, este objeto deve implementar uma interface de notificação. O trecho de código adiante exemplifica o registro de interesse e a interface de notificação, ambos relacionados à falha de um objeto JIC:

```
public class MyGridClient
                    implements FailureInterestedEventProcessor{
  ...
  mgID = "john@frodo.lsd.ufcg.edu.br/myGridAP/mygrid/1";
  FailureDetector fd = accesspoint.getFailureDetector();
  fd.registerFailureInterested(this, mgID);
  ...
  public void notifyFailure(EventProcessor ep, double conf) {
    System.out.println("EventProcessor " + ep + "Falhou");
  }
}
```

Note que é possível recuperar o detector de falhas através do *Access Point*, e logo depois, usar o detector de falhas para registrar interesse na falha do objeto cuja identificação é representada por `mgID`. É importante reparar que, no momento do registro, a instância da classe `MyGridClient` é passada como primeiro parâmetro. Este parâmetro representa a entidade interessada. O segundo parâmetro representa a identificação da entidade de interesse. Existem sobrecargas deste método que permitem passar a própria referência do objeto de interesse ao invés de sua identificação. Note também que o objeto interessado precisa implementar a interface `FailureInterestedEventProcessor`, que define o método `notifyFailure`.

Uma vez que o registro é realizado, o módulo de detecção de falhas começa a monitorar o objeto de interesse. Caso o módulo do *Access Point* local detecte a falha do objeto de interesse, o método `notifyFailure` da entidade interessada será executado, ou seja, o

detector notifica o objeto interessado sobre a falha do objeto.

Um detalhe interessante está no fato de que o tratamento de falhas de objetos distribuídos, para cada classe, está centralizado em um único método, evitando que o tratamento de falhas seja realizado de maneira espalhada no código. Isso facilita manutenção e evita duplicação de código. Além disso, o programador adquire a disciplina de pensar em sua aplicação como uma máquina de estados. Ao receber uma notificação sobre a falha de um determinado objeto, o programador é levado a pensar sobre o estado de sua aplicação antes de tomar a decisão diante da falha. Consideramos esta uma boa prática, uma vez que é comum que programadores codifiquem suas aplicações distribuídas pensando apenas nas situações de sucesso. Apenas depois de vários testes e *bugs*, o programador é levado a pensar em algumas situações de falha. Portanto, acreditamos que o JIC ajuda o programador a pensar em falhas e organizar seu raciocínio sobre as situações em que elas acontecem.

Uma vez notificado, o cliente JIC tem seu interesse automaticamente descadastrado. Porém, o cliente pode optar por desregistrar o interesse em qualquer momento, como mostra o trecho de código abaixo:

```
...
mgID = "john@frodo.lsd.ufcg.edu.br/myGridAP/mygrid/1";
FailureDetector fd = accesspoint.getFailureDetector();
...
fd.unregisterFailureInterested(this, mgID);
...
```

Além das interfaces apresentadas acima, o JIC fornece interfaces correspondentes para registrar e desregistrar interesse na recuperação de um serviço, bem como tratar a notificação de uma recuperação. Neste caso, um *Event Processor* deve implementar a interface `RecoveryInterestedEventProcessor`, que define o método `notifyRecovery`, além de usar os métodos do detector de falhas `registerRecoveryInterested` e `unregisterRecoveryInterested` para registrar e desregistrar, respectivamente, interesse na recuperação de um serviço.

4.5 Implementação

Ao produzir um software, existem diversos aspectos que não são contemplados pelo projeto arquitetural. Estes aspectos, em geral de mais baixo nível, costumam receber a devida atenção durante a fase de implementação. Apesar de não aparecerem na arquitetura, muitos destes aspectos são relevantes para que se entenda o comportamento da aplicação em questão. Aqui, abordaremos os detalhes de implementação que julgamos mais relevantes.

4.5.1 Linguagem de Programação

Desde o início, o JIC foi projetado para ser dependente de linguagem de programação, no caso, Java. Portanto, o modelo de objetos remotos no JIC foi construído sobre o modelo de objetos Java. De fato, soluções como CORBA, devido a um requisito de interoperabilidade, consideram um modelo de objetos neutro em relação à linguagem de programação. Como consequência, estas soluções não atingem a integração desejada com a linguagem [WRW96], apesar de ser possível adaptá-las para lidar com objetos Java, por exemplo. O JIC, por sua vez, busca manter a maior proximidade possível em relação ao modelo de objetos Java, tornando explícitas, porém, as diferenças impostas pela computação distribuída.

No entanto, o fato de estarmos usando Java, mais precisamente, a maneira que estamos usando Java, impõe pelo menos uma dependência séria em relação à linguagem. Esta dependência é decorrente do uso do mecanismo de "serialização" de Java para enviar eventos entre *Access Points*. Uma vez que o JIC usa Jabber como infra-estrutura para troca de mensagens e o protocolo Jabber realiza troca de mensagens XML (i.e., há neutralidade em relação à linguagem de programação), não faz sentido usar o mecanismo de "serialização" para enviar mensagens entre dois *Access Points*. Contudo, uma das decisões de implementação consistiu em usar a biblioteca *smack* [Jiv06] para se comunicar com servidores Jabber. Esta biblioteca, além de ser escrita em Java, foi considerada a biblioteca de *código aberto* mais confiável dentre as que analisamos e a atividade da comunidade em relação ao software é bastante intensa. Esta biblioteca, porém, "serializa" os objeto Java que serão enviados para o lado remoto através do mecanismo fornecido pela linguagem e os encapsula nas mensagens XML. Logo, o lado remoto precisa conhecer o mecanismo de "serialização" usado por Java para "desserializar" os objetos recebidos.

Todavia, não acreditamos que haja obstáculos que, no futuro, nos impeçam de usar XML para codificar as mensagens do JIC sem usar a "serialização" de Java. Esta seria uma maneira neutra da codificar as mensagens e obter independência em relação à linguagem. Entretanto, existe o argumento de que para alcançar real interoperabilidade, é preciso codificar manualmente o XML que representa as mensagens de uma aplicação [LS05], ou seja, este parece ser o preço para realmente se ter interoperabilidade entre linguagens.

4.5.2 *Stubs* e Eventos

A principal entidade que torna possível a integração com a linguagem de programação é o *stub*. Um *stub* é a representação local do objeto remoto com o qual se pretende comunicar. Por isso, apresenta a mesma interface oferecida pelo objeto remoto que referencia. *Stubs* conhecem detalhes de baixo nível da infra-estrutura de comunicação, sendo então responsáveis por traduzir as invocações recebidas em mensagens e encaminhar tais mensagens para os objetos remotos correspondentes. Cada *stub*, ao ser criado, recebe uma identificação (JID), que indica o objeto remoto por ele representado. Também no momento da criação, o *stub* recebe uma referência para uma camada interna do *Access Point*, chamada `CommunicationLayer`. Esta camada representa a fronteira entre o mundo orientado a objetos e o mundo Jabber, onde se usa primitivas *send/receive* para troca de mensagens. Ao receber uma invocação, o *stub* reúne as informações sobre o método invocado, as encapsula em um `JICEvent` e o repassa para a `CommunicationLayer`, que providenciará o seu envio através de mensagens Jabber. Um `JICEvent` é uma estrutura de dados interna do JIC formada pelos campos apresentados na Figura 4.8:

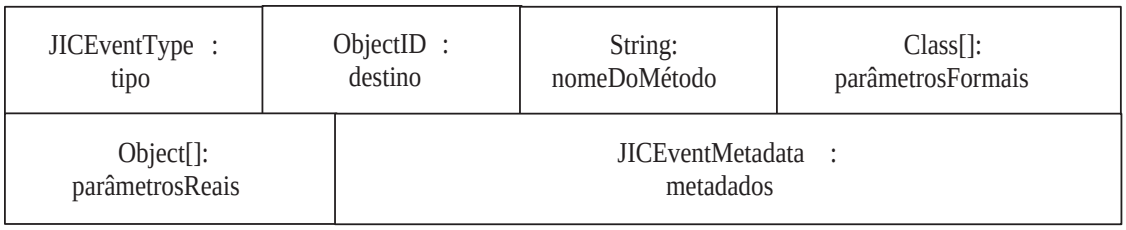


Figura 4.8: Campos de um evento JIC.

O primeiro campo representa o tipo do evento. Basicamente, serve para diferenciar três tipos de eventos: (i) eventos de aplicação, (ii) eventos de detecção de falhas e (iii) eventos

de controle. Eventos de aplicação representam invocações em objetos remotos da aplicação. Já os eventos de detecção de falhas representam invocações remotas que são executadas nos módulos de detecção de falhas de cada *Access Point*. Por fim, eventos de controle são interceptados pela `CommunicationLayer` e usados para realizar a negociação do protocolo de conexão.

O próximo campo (destino) é um JID que identifica o objeto no qual a invocação será executada. Adiante, os próximos dois campos armazenam as coleções de parâmetros formais e reais da invocação. Note que estas duas coleções são relacionadas, de forma que o elemento i da coleção de parâmetros formais determina o tipo do objeto i da coleção de parâmetros reais. Por fim, temos um campo com metadados, representado por um objeto `JICEventMetadata`. Este objeto existe para prover flexibilidade ao JIC. Sendo assim, é possível adicionar novos campos a um evento no futuro, sem perder compatibilidade com versões anteriores do código. Atualmente, o JIC usa metadados para acrescentar aos eventos os campos mostrados na Figura 4.9

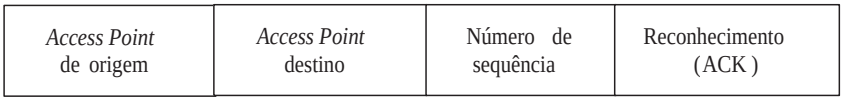


Figura 4.9: Campos de um `JICEventMetadata`

Atualmente, todos estes campos presentes como metadados são usados nos eventos de controle pelo protocolo de conexão. Os dois primeiros representam as identificações dos *Access Points* de origem e destino, respectivamente. Em seguida, temos o número de sequência, também usado nos eventos de aplicação, o que permite que eventos perdidos sejam identificados. O último campo armazena o reconhecimento dos números de sequência recebidos durante a negociação de uma conexão.

Ao ser recebido por um *Access Point*, um evento é colocado na fila de invocações, conforme explicado na Seção 4.1.1. Quando um *Event Handler* remove este evento da fila, a primeira coisa a ser feita é interpretar os campos do `JICEvent`. Com as informações sobre o nome do método, parâmetros reais e formais e a própria identificação do *Event Processor* alvo, o *Event Handler* usa a API de reflexão de Java [Mic06] para invocar o método neste *Event Processor*. É importante ressaltar que, se o evento estiver endereçado para um *Event Processor* que não está exportado, o tratamento do erro que acontece no *Access Point* destino

é silencioso, ou seja, a invocação é descartada e nenhuma entidade é notificada de que algum erro aconteceu. Isto faz sentido porque, se o objeto não está exportado, as mensagens de monitoração receberão respostas de que o objeto não está disponível, de forma que o cliente será notificado da falha do objeto e não poderá considerar que sua invocação aconteceu. Caso o cliente não esteja interessado na falha do objeto, isso significa que a comunicação acontece no modelo *fire-and-forget*, no qual um cliente dispara invocações, mas não tem interesse em saber se elas realmente serão atendidas.

Um `JICEvent` representa o componente do JIC que é trocado entre duas entidades remotas, visto que encapsula toda comunicação realizada através da infra-estrutura JIC. Na medida em que o software evolui, entidades remotas podem estar usando versões diferentes do JIC. Contudo, é importante que a infra-estrutura de comunicação mantenha compatibilidade para trás, evitando que aplicações deixem de se comunicar devido a mudanças na infra-estrutura de comunicação. Por isso, ao longo das novas versões do JIC, deve-se manter compatibilidade no código do `JICEvent`. Atualmente, fazemos isso através do próprio esquema de versionamento presente no mecanismo de serialização de Java. Um `JICEvent` mantém um campo `SerialVersionUID`, com a versão da classe, indicando que esta versão está sendo controlada explicitamente no código. Alterações no código que não modificam a hierarquia de classes (na qual o mecanismo de serialização se apóia) podem ser realizadas em um `JICEvent` sem comprometer compatibilidade. Diante dessas possíveis mudanças, é preciso manter o `SerialVersionUID` inalterado para preservar a compatibilidade em versões futuras.

4.5.3 Semântica das Invocações

Como visto na Seção 4.4.2, a classe `SimpleEventProcessor` reescreve os métodos `equals`, `toString` e `hashCode` da classe `Object`, especializando as semânticas destes métodos para ambientes distribuídos.

Através da nova semântica, a comparação de objetos remotos considera referências e não comparação de conteúdo. Comparar conteúdo significaria realizar invocações remotas, o que complicaria a semântica, uma vez que o JIC não oferece suporte à invocações com retorno. Portanto, para preservar compatibilidade de interfaces com o modelo de objetos locais de Java, a execução do método `equals` verifica apenas a identificação do objeto remoto. Caso

a identificação dos objetos em comparação seja a mesma, ambas as referências apontam para o **mesmo** objeto remoto, de forma que o método `equals` (executado localmente no *Event Processor* ou *stub*) retorna `true`.

De maneira análoga, o método `hashCode` retorna valores iguais para objetos que referenciam um mesmo objeto remoto. Já o método `toString` retorna a identificação do objeto, ou seja, a representação textual de sua referência remota. Esta identificação é o próprio JID do objeto referenciado. `hashCode` e `toString` também são executados localmente, assim como o método `equals`.

4.5.4 *CommunicationLayer*

Na Seção 4.5.2, vimos que o *stub* cria um evento e o repassa para a `CommunicationLayer`. Esta, por sua vez, usa o objeto de comunicação apresentado na Seção 4.1 para encapsular o evento em mensagens Jabber e enviá-lo para o *Access Point* destino. Porém, a `CommunicationLayer` oferece flexibilidade para a manipulação de eventos antes que sejam enviados através do Jabber. Suponha que o desenvolvedor queira aplicar um algoritmo de compressão antes de enviar um evento, ou usar um esquema de criptografia, por exemplo. A `CommunicationLayer` permite realizar tal manipulação através de uma cadeia de filtros, usada durante o envio e recebimento de `JICEvents`. A `CommunicationLayer` é formada por um conjunto de filtros, seguindo o padrão *cadeia de responsabilidades (chain of responsibility)* [GHJV94], conforme mostrado na Figura 4.10. No momento do envio, o último filtro da cadeia é o objeto de comunicação que, naturalmente, representa o primeiro filtro da cadeia no momento de recebimento.

A `CommunicationLayer` oferece a possibilidade de adicionar filtros à cadeia existente. Através do método `addFilter`, um novo filtro é adicionado entre o `ApplicationFilter` e o filtro que ele referencia. Na Figura 4.10, um novo filtro A seria inserido entre o `ApplicationFilter` e o `ConnectionFilter`. Logo após essa inserção, um novo filtro B seria inserido entre o `ApplicationFilter` e o filtro A, e assim por diante. Vale salientar que, um filtro possui duas funcionalidades, sendo uma no momento do envio e outra no momento do recebimento. Tais funcionalidades devem ser complementares, ou seja, um filtro que realiza compressão no momento de envio, deve realizar descompressão no momento de recebimento. Note também que a cadeia de filtros no momento do recebi-

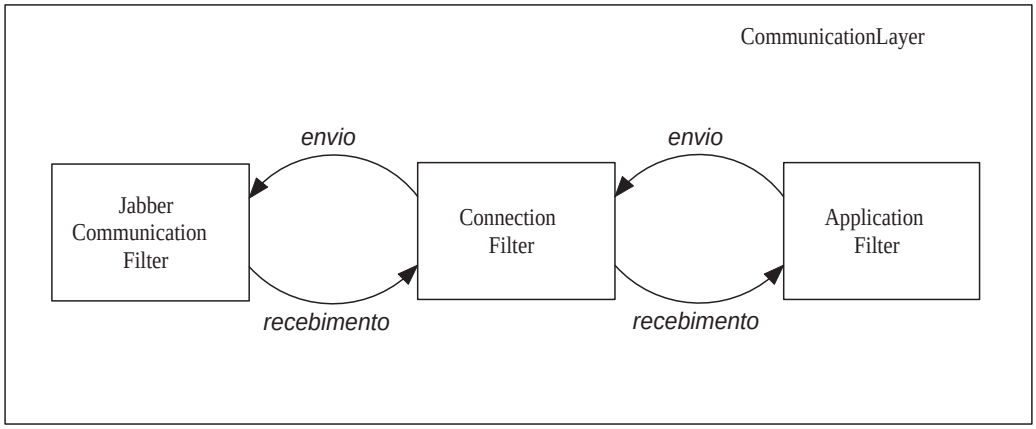


Figura 4.10: Padrão *cadeia de responsabilidades* implementado pela `CommunicationLayer`.

mento é processada em ordem inversa ao processamento realizado durante o envio. Com exceção dos filtros originais implementados no JIC, um filtro previamente adicionado pode ser removido através do método `removeFilter()` da `CommunicationLayer`.

Implementar um filtro é uma tarefa bastante simples: basta estender a classe abstrata `Filter` e implementar o comportamento dos métodos abstratos `onSend(JICEvent)` e `onReceive(JICEvent)`. Estes métodos são usados para manipular um evento durante o envio e recebimento do mesmo, respectivamente. A classe abstrata fornecida pelo JIC já implementa o código necessário para realizar o comportamento do padrão "cadeia de responsabilidades". Um outro detalhe importante no esquema de filtros diz respeito ao tratamento de erros. Em qualquer momento, os métodos `onSend(JICEvent)` e `onReceive(JICEvent)` podem lançar uma exceção do tipo `FilterException`. Esta exceção interrompe o fluxo na cadeia de filtros, além de ativar um mecanismo de notificação do JIC, em que objetos interessados (`FilterExceptionListeners`) são notificados de que a exceção aconteceu. Este esquema de notificação segue o padrão observer [GHJV94]. Por exemplo, considere o filtro que negocia a conexão e verifica o número de sequência dos eventos de aplicação. Caso o número de sequência observado seja maior que o esperado, ou seja, pelo menos um evento foi perdido, o filtro lança uma exceção `EventLostException` que estende `FilterException`. O módulo detector de falhas estará cadastrado como *listener* de exceções na cadeia de filtros e poderá tomar as decisões necessárias, como identificar o *Access Point* de origem da mensagem perdida e suspeitar de

seus objetos.

Note que o esquema de filtros pode facilitar o desenvolvimento de testes para a aplicação. Uma prática bastante comum em testes de sistemas distribuídos consiste em injetar falhas nos seus componentes [Ram04; DJMT96]. Em geral, é muito comum injetar falhas de omissão e falhas por parada (fail-stop). Falhas por omissão são caracterizadas pela perda de mensagens, seja durante o processamento no sistema operacional de um processo ou por problemas na infra-estrutura de comunicação (e.g., mensagens descartadas pela rede). A situação de falha por parada é caracterizada pelo fim da execução de um processo, de forma que esta parada pode ser detectada por outros componentes do sistema. Com o JIC, através do padrão usado na `CommunicationLayer`, é possível inserir filtros que facilmente provocam estes tipos de falha. Por exemplo, pode-se inserir um filtro que descarta o *enésimo* evento enviado/recebido, ou ainda que descarta aleatoriamente uma porcentagem dos eventos enviados/recebidos. Note que este cenário representa exatamente situações de falha por omissão. Pode-se também inserir um filtro na camada de comunicação que, depois de algum tempo (ou de alguns eventos trocados), descarta todos os eventos que deveriam ser enviados ou recebidos por um objeto, caracterizando um cenário de falha por parada, mesmo que o objeto ainda esteja exportado. Note que outros tipos de falha também podem ser injetadas facilmente, como por exemplo, falhas bizantinas, em que um objeto se comporta de maneira arbitrária, desviando-se do comportamento esperado. Para reproduzir um comportamento diferenciado, basta inserir um filtro que modifique o conteúdo dos eventos.

4.5.5 Protocolo de Conexão

Dentre os filtros originais implementados pelo JIC, mencionamos anteriormente o `ConnectionFilter`, responsável por estabelecer conexão entre dois *Access Points* e gerenciar os números de sequência dos eventos de aplicação (Conforme explicado na Seção 4.3). Através de número de sequência, o JIC pode identificar mensagens perdidas entre *Access Points*, podendo assim fornecer a semântica apresentada na Seção 4.3. Para negociar a conexão, o `ConnectionFilter` implementa o protocolo descrito em [Tom75]. Basicamente, temos a negociação em três vias apresentada na Figura 4.11:

A situação típica de negociação é mostrada na Figura 4.11(a). Nesta situação, uma entidade A tenta iniciar conexão com uma entidade B. Inicialmente, A envia uma mensagem

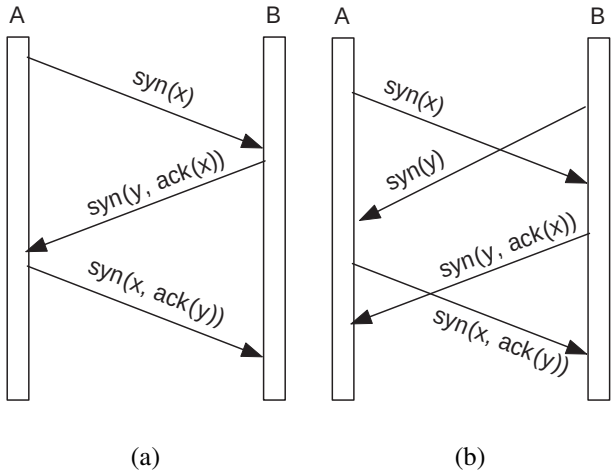


Figura 4.11: (a) Negociação da conexão iniciada pelo objeto A e (b) iniciação simultânea.

de sincronização (`SYN`) para a B, passando o valor do número de sequência que pretende usar (X). O objeto B, ao receber o `SYN`, não sabe se recebeu realmente uma mensagem de sincronização válida, ou se recebeu uma duplicata de alguma mensagem antiga. B então responde a mensagem recebida com um (`SYN`), passando o valor que pretende usar como número de sequência (Y) e também envia um reconhecimento (`ACK`), confirmando o valor X recebido. Ao receber o `SYN+ACK`, A sabe que está recebendo uma mensagem atual, pois esta mensagem possui o reconhecimento do valor X que A enviou no primeiro `SYN`. Por isso, A aceita o valor Y que B enviou como sendo o número de sequência escolhido por B. Dessa forma, A já considera que a conexão está estabelecida. Porém, B ainda não sabe se o `SYN` recebido inicialmente é válido. Por isso, A responde o `SYN+ACK` de B reafirmando o seu número de sequência e reconhecendo o número de sequência Y enviado por B. Ao receber o reconhecimento, B tem a certeza de que o `SYN` enviado por A não é uma duplicata de uma mensagem antiga e cosidera a conexão como estabelecida. A partir deste momento, cada mensagem enviada de B para A e vice-versa, terão número de sequência encapsulado, seguindo incremento unitário a cada mensagem. As entidades podem, a qualquer momento, executar um protocolo de *release* com negociação em três vias para fechar a conexão, invalidando os números de sequência em uso.

A Figura 4.11(b) representa o mesmo processo de negociação, porém na situação em que ambas as entidades tentam iniciar a conexão no mesmo momento (ou quase no mesmo momento), caracterizando um cenário conhecido como “iniciação simultânea”. Neste caso,

cada entidade, após enviar um `SYN`, com seu valor para número de sequência, recebe um outro `SYN`, porém sem qualquer `ACK`. Isso significa que, no momento em que cada entidade esperava uma resposta, recebeu-se uma nova requisição. Diante deste cenário, cada entidade envia um `SYN` reafirmando o valor escolhido e reconhece o valor recebido através de um `ACK`. Ao receber o `SYN` + `ACK`, cada entidade sabe que a outra recebeu o seu valor (pois o está reconhecendo) e que o valor recebido no `SYN` confirma a primeira mensagem, podendo ser aceito. Neste momento, cada entidade considera que a conexão está estabelecida. Processo semelhante existe para negociar o fechamento de uma conexão, quando ambas as entidades o tentam fazer simultaneamente.

Implementar um filtro para realizar este gerenciamento de conexões pareceu simples de início. O único cuidado extra consistiu em usar um mecanismo de retransmissão, caso as mensagens de controle (`SYN`s e `ACK`s) fossem perdidas. Quando uma entidade espera uma resposta a um `SYN`, por exemplo, e esta resposta não é recebida dentro de um *timeout*, a entidade retransmite a última mensagem, pois a outra entidade pode não tê-la recebido. Depois de um determinado número de retransmissões (valor padrão usado pelo TCP, mas que pode ser configurável), a entidade desiste de retransmitir e abandona o processo de negociação.

4.5.6 Detecção de Falhas

Na seção anterior, vimos que o filtro de conexão adiciona número de sequência apenas aos eventos de aplicação. De fato, eventos usados pelo mecanismo de detecção de falhas não precisam carregar número de sequência. Em primeiro lugar, porque não queremos suspeitar de objetos de aplicação caso um evento de detecção seja perdido. Além disso, o esquema de detecção é robusto o suficiente para manter-se funcionando quando uma mensagem é perdida (veja Seção 2.4.1).

O mecanismo de detecção de falhas do JIC é implementado através de um *framework* que usa as idéias e conceitos apresentados em [FDGO99]. Neste trabalho, detectores de falhas são considerados objetos de primeira ordem em um sistema distribuído, conforme definido na Seção 2.4.1.

No framework implementado sobre o JIC, optamos por usar um monitoramento remoto baseado no modelo *pull*. Porém, os objetos `Notificáveis` são informados sobre mudanças de estado através de um modelo *push*. Inicialmente, implementamos o modelo probabi-

lístico definido em [HDYK04], explicado na Seção 2.4.1.

Contudo, enfrentamos alguns problemas com esta abordagem quando a colocamos em um ambiente interno de produção. Em primeiro lugar, observamos um comportamento instável antes da amostra atingir o valor de n (realizamos experimentos com $n = 100$). "Comportamento instável" significa uma alta taxa de falsas suspeições (λ_M) devido à oscilação na média e desvio padrão da amostra, uma vez que a amostra era muito pequena. Este problema era esperado, uma vez que os resultados em [HDYK04] mostram que quanto menor o tamanho da amostra, maior o valor de λ_M. Portanto, usar o modelo de detecção sem ter a janela completa implica muitos erros de detecção no início do processo, independente da qualidade de serviço escolhida pela aplicação. Isso significa que, para aplicações que possuem demanda por baixa valor de λ_M, o uso deste modelo sem que a amostra esteja completa provavelmente irá de encontro a esta demanda, não a satisfazendo. Por outro lado, pode-se postergar o início do processo de detecção para quando a janela estiver completa, evitando este comportamento instável. Mais uma vez, esta alternativa pode afetar diretamente a qualidade de serviço da detecção. Caso uma aplicação tenha demanda por baixo tempo de detecção (T_D) e o tamanho da janela estiver definido com um valor muito alto, o tempo de preenchimento da janela pode demorar muito (este tempo também depende do período de envio de *heartbeats*), fazendo com que a qualidade de serviço seja violada.

A solução temporária que adotamos consiste em, durante o período de preparação, usar um mecanismo de detecção que procure se manter o mais próximo possível da qualidade de serviço demandada pela aplicação. De acordo com esta qualidade, um período de envio de *heartbeats* (Δ_i) e um *timeout* estático Δ_{to} podem ser calculados. Sendo assim, um mecanismo simples, porém pouco flexível, pode ser usado, no qual o `Monitor` solicita *heartbeats* ao `Monitorável` em intervalos de Δ_i e espera receber uma resposta dentro do período Δ_{to}, suspeitando do `Monitorável` caso não receba nesse período. Neste mecanismo, Δ_i e Δ_{to} são estáticos. Terminado o período de preparação, muda-se o mecanismo para o probabilístico [HDYK04], agora com a janela completa. Apesar de parecer uma solução razoável, não temos como garantir qualidade de serviço se aparecem mudanças no ambiente (e.g., maior carga na rede) durante o período de preparação, pois a solução usada neste período é estática. Além disso, vale salientar que, quando uma entidade falha e aparece novamente, a janela anterior não serve como amostra, sendo necessário um novo período de

preparação.

Todavia, o problema apresentado acima não é o único. Observamos em nossas baterias de execuções que existem situações nas quais o mecanismo de detecção de falhas se torna bastante sensível a variações sutis nos tempos de chegada de *heartbeats*. Para ilustrar esta situação, apresentamos um exemplo bem simples. Considere uma aplicação composta por dois objetos, A e B, exportados nos Access Points AP_A e AP_B respectivamente. Ambos os *Access Points* estão executando em uma mesma máquina, porém em espaços de endereçamento diferentes. A máquina está dedicada à execução desta aplicação. Após a exportação de ambos os objetos, A se interessa na falha de B. O detector de falhas de AP_A começa então a monitorar a B, enviando mensagens de monitoração para AP_B. O módulo de detecção de AP_B responde as solicitações de informação através de *heartbeats*. Uma vez que não há carga extra na máquina, o tempo entre solicitações se mantém constante e a comunicação é feita via *loopback* (não usa a rede). Portanto, os *heartbeats* chegam em intervalos de tempos praticamente constantes. Com isso, a janela é completamente preenchida com uma distribuição cujo desvio padrão é muito pequeno, uma vez que todos os *heartbeats* chegam em intervalos muito próximos da média amostral. Dessa maneira, qualquer carga mínima exercida no ambiente de execução (e.g., o movimento repetido do mouse), faz com que algum *heartbeat* se atrase poucos milésimos de segundo, o que representa uma variação muito grande na distribuição. Como resultado, A suspeita de B, mesmo que B não tenha falhado. Concluímos que o comportamento do mecanismo de detecção não está errado, mas está sendo usado em um ambiente cuja distribuição dos intervalos de chegada dos *heartbeats* não é uma distribuição normal, como considera o modelo de detecção apresentado em [HDYK04]. De fato, este problema foi verificado em várias configurações (diferentes combinações de tamanho de janela, Δ_i, máquinas diferentes, etc), e foi observado em muitas das execuções em redes locais. Note que, apesar de o ambiente usado como exemplo (uma única máquina) não ser o ambiente alvo, O JIC precisa funcionar neste ambiente também. Por exemplo, é nele que o desenvolvedor realiza seus primeiros testes, de forma que o comportamento deveria ser semelhante ao verificado no ambiente de produção.

Diante destas situações, acreditamos que a resolução destes problemas representa um assunto em aberto, ou seja, uma oportunidade para pesquisa futura, que não pertence ao escopo desta dissertação. A solução temporária que escolhemos consiste em adotar apenas

o mecanismo estático que usamos antes de preencher a janela do modelo probabilístico. Permitimos, no entanto, que o usuário forneça valores para o Δ_i e para Δ_{to}. Todavia, deixar a escolha destes parâmetros para o usuário não garante que a solução será bem sucedida, uma vez que o usuário em geral não sabe quais valores escolher e a escolha depende do ambiente em que a aplicação é executada. Para tornar mais fácil e mais precisa a escolha destes parâmetros, sugerimos a construção de uma simples aplicação de teste que executa no ambiente alvo durante alguns dias. Esta aplicação de teste realiza um experimento através dos seguintes passos:

1. Primeiramente, o usuário define a quantidade de banda que deseja usar para detecção de falhas, que será denotada por L_b;

2. Da mesma forma, o usuário define a quantidade média de entidades que serão monitoradas simultaneamente em um dado momento, que representaremos por n;

3. Através destes valores, é possível calcular o valor mais adequado para o Δ_i, aproveitando bem a banda e sem ultrapassar seu limite máximo. Para calcular este valor, usaremos algumas outras variáveis. Uma vez que a detecção de falhas acontece através de um modelo *pull*, considere t_s como o tamanho da mensagem de solicitação enviada a um `Monitorável` e t_h como tamanho de um *heartbeat* enviado em resposta a esta mensagem de solicitação. Assim, o consumo de banda pelo mecanismo de detecção de falhas (B_{df}) será:

$$B_{df} = n \cdot (t_s + t_h) \tag{4.1}$$

   A equação 4.1 representa a quantidade de banda consumida em uma rodada de detecção.

4. Podemos então definir Δ_i em função da banda disponível e da banda utilizada pelo mecanismo de detecção de falhas:

$$\Delta_i = \frac{B_{df}}{L_b} \tag{4.2}$$

Isso significa que a banda usada em uma rodada de detecção de falhas dividida pela banda disponível determina o intervalo entre solicitações de *heartbeats*, de forma que o consumo médio de banda seja no máximo L_b.

5. Fixado o Δ_i, o experimento executa por alguns dias, solicitando *heartbeats* a cada Δ_i unidades de tempo;

6. Finalizada a medição, o experimento capturou as características do ambiente, ou seja, a distribuição de chegada dos *heartbeats* (rtt, i.e., *round-trip time* entre o envio da mensagem de solicitação e a chegada do *heartbeat* correspondente). Sendo assim, o *timeout* de detecção Δ_{to} deve ser escolhido de acordo com esta distribuição. Note que, se Δ_{to} possuir valor maior que o rtt em um dado instante α ($\Delta_{to} > rtt_\alpha$), teremos uma falsa suspeição. Logo, o valor de Δ_{to} influencia diretamente na taxa de falsas suspeições λ_M. Por isso, considerando a distribuição de rtts observada através do experimento, o usuário pode calibrar o valor de Δ_{to} e estabelecer a probabilidade de haver uma falsa suspeição ($P(rtt > \Delta_{to})$). Existe um *trade-off* entre tempo de detecção e taxa de falsas suspeições. Quanto maior o Δ_{to}, menor λ_M, mas o tempo de detecção T_D aumenta. Na medida em que Δ_{to} diminui, λ_M aumenta, mas T_D diminui.

7. Escolhido um Δ_{to}, é possível calcular o tempo de detecção de falhas máximo (T_{Dmax}). Considere que a banda é simétrica (uma mensagem é enviada do `Monitor` para o `Monitorável` em tempo $rtt/2$). No pior caso, um `Monitorável` falha imediatamente após enviar um *heartbeat* (aproximadamente $rtt/2$ unidades de tempo depois de o `Monitor` ter enviado a solicitação). O `Monitor` recebe o *heartbeat* e realiza uma nova solicitação (Δ_i unidades de tempo depois da primeira, ou seja, $\Delta_i - (rtt/2)$ unidades de tempo após a falha do `Monitorável`). Depois de esperar durante o *timeout* Δ_{to}, o detector de falhas do `Monitor` irá suspeitar do `Monitorável`. Logo, T_{Dmax} é definido pela fórmula:

$$T_{Dmax} = \Delta_i - \frac{\overline{rtt}}{2} + \Delta_{to} \tag{4.3}$$

Nesta fórmula, $\overline{rtt}$ é a média da distribuição de rtts. Através de tal fórmula 4.3, é possível ajustar tempo de detecção e *timeout*, balanceando tempo de detecção e taxa

de falsas suspeições.

Logo, através deste experimento, o usuário poderá preencher um arquivo de configuração de sua aplicação com os valores de Δ_i e Δto, calibrando o detector de falhas de acordo com a qualidade de serviço desejada e com as características do ambiente alvo da aplicação.

Capítulo 5

Avaliação da Solução

Neste capítulo, iremos avaliar o JIC sob duas perspectivas. A primeira diz respeito a *desempenho*. É desejável que a solução proposta possua desempenho pelo menos comparável às soluções usadas nas situações em que o JIC se propõe a ser usado. A segunda considera alguns *aspectos relacionados a Engenharia de Software*. Consiste em avaliar as implicações positivas e negativas na produção de aplicações distribuídas baseadas em JIC. Dentre tais implicações, esperamos benefícios decorrentes do escopo bem definido para as threads de uma aplicação. Estas consequências positivas se devem ao fato de ser possível para o programador desenvolver aplicações distribuídas sem se preocupar com os problemas inerentes ao modelo de threads, ou ainda limitar o escopo sobre o qual deverá raciocinar em relação a concorrência. Em ambas as avaliações, o JIC será comparado com Java RMI, que é um exemplo representativo de *middleware orientado a objetos*, sendo bastante usado e bem sucedido.

Na primeira avaliação, observamos que o JIC possui um desempenho comparável a Java RMI. Em relação aos aspectos de Engenharia de Software avaliados, também observamos implicações positivas. Dessa forma, consideramos que o uso de JIC para o desenvolvimentos de aplicações distribuídas não-bloqueantes compensa, pois conseguimos minimizar os problemas observados em objetos distribuídos, sem perdas significativas de desempenho.

5.1 Avaliação de Desempenho

A avaliação de desempenho foi realizada sobre uma simples aplicação de multiplicação de matrizes. Basicamente, a aplicação lê duas matrizes do disco rígido, multiplica a primeira pela segunda e retorna o maior valor dentre as somas dos elementos de cada coluna da matriz resultante. A leitura das matrizes acontece de maneira aleatória em um arquivo de entrada local que possui um milhão de valores do tipo *double* de Java (64 bits). Assim, evita-se que a leitura esteja sempre em *cache* depois de algumas invocações. Uma requisição é composta por um número inteiro, que representa as ordens das matrizes que serão multiplicadas. Para cada requisição, a entidade que a recebe executa o procedimento de leitura e multiplicação duas vezes. Na primeira execução, existe controle de concorrência através de um bloco *synchronized*, ou seja, existe uma região crítica cujo acesso é restrito a uma única *thread* através de um mecanismo de exclusão mútua. Na segunda execução, não existe controle de concorrência no código. O maior valor dentre as duas execuções é retornado como resultado. Acreditamos que esta aplicação é um exemplo representativo de aplicações reais, uma vez que combina processamento e acesso a disco, além de combinar trechos de código em que há regiões críticas com trechos em que a memória é compartilhável.

5.1.1 Experimento

O experimento foi executado na rede local do Laboratório de Sistemas Distribuídos (LSD) da Universidade Federal de Campina Grande (UFCG), em máquinas que possuem aproximadamente as mesmas configurações (Pentium IV, com 1 gigabyte de memória RAM, executando o Sistema Operacional Linux Debian Sarge). Todas as máquinas estavam configuradas com a mesma versão da máquina virtual Java (versão 1.5). Os experimentos foram executados em máquinas dedicadas, de forma que as medidas de tempo não foram influenciadas por outros processos. Além disso, as execuções foram conduzidas em períodos de baixo tráfego na rede (durante a semana à noite e em finais de semana), na intenção de evitar influências nos resultados devido à contenção de recursos de comunicação.

Dividimos o experimento em dois cenários, de maneira que um abrange aplicações *cliente-servidor* e outro contempla aplicações *peer-to-peer*. No primeiro, aplicações clientes elaboram requisições para um único servidor, especificando as ordens das matrizes a

serem multiplicadas. O servidor processa as multiplicações e retorna o resultado, delimitando o fim da requisição. No cenário *peer-to-peer*, os vários *peers* estão conectados em clique. Cada *peer* elabora requisições da mesma forma que no modelo cliente-servidor. No entanto, ao invés de enviá-las para um servidor único, cada *peer* encaminha suas requisições para três outros *peers* aleatórios, que processam as requisições e as repassam aleatoriamente para outros três *peers*. Ao receber uma mesma requisição pela segunda vez, um *peer* apenas retorna o valor processado anteriormente e não a repassa para os demais. Ao receber uma resposta referente a uma requisição que foi encaminhada, o *peer* que a encaminhou compara o resultado recebido com o resultado local e mantém o maior valor dentre estes. Portanto, ao receber todas as respostas, o maior valor dentre todos os recebidos por um *peer* será mantido. Este valor é retornado ao *peer* que encaminhou a requisição e o processo continua até que o resultado retorne para o *peer* que a criou.

Para cada cenário, executamos duas baterias de experimentos, apenas modificando a granularidade das requisições. Tal granularidade é determinada pelas ordens das matrizes a serem multiplicadas. Na primeira bateria, as requisições solicitavam o processamento de matrizes de ordem pequena (ordem 10), cujo processamento acontecia em poucos milésimos de segundo. Na segunda bateria, as requisições assumiam granularidade maior (ordem 150), cujo processamento acontecia na ordem de alguns segundos. Em cada execução, a aplicação efetuada por cada cliente/*peer* consiste na solicitação de 100 requisições. Em todos os cenários, avaliamos o desempenho da mesma aplicação implementada sobre JIC e sobre Java RMI.

Para cada bateria de experimentos do cenário cliente-servidor, foi feita uma análise incremental em relação ao número de clientes. Mantivemos fixo o número de servidores (apenas um) e variamos a quantidade de clientes (um, três, cinco, dez e vinte). Através de tal variação, é possível analisar o comportamento do servidor nas situações em que existem acessos simultâneos ao mesmo, situações estas típicas de sistemas distribuídos.

As comparações de desempenho entre a aplicação que usa RMI e a aplicação que usa JIC consideraram execuções idênticas. No cenário *peer-to-peer*, em que cada *peer* roteia requisições para outros três, é preciso considerar as decisões de roteamento ao analisar se duas execuções são idênticas. Seja E uma execução, P o conjunto de peers que participam desta execução, Req o conjunto de requisições efetuadas por cada *peer* e Rot o conjunto de

decisões de roteamento de cada *peer*, dizemos que uma execução E_1 é idêntica à execução E_2, se P_1 = P_2, Req_1 = Req_2 e Rot_1 = Rot_2. Para garantir que as execuções dos experimentos seriam idênticas, cada *peer* possuía um arquivo contendo a sequência de decisões de roteamento, considerando todos os outros *peers*. Então, os arquivos que foram usados para cada *peer* durante o experimento com RMI foram mantidos para o experimento com o JIC.

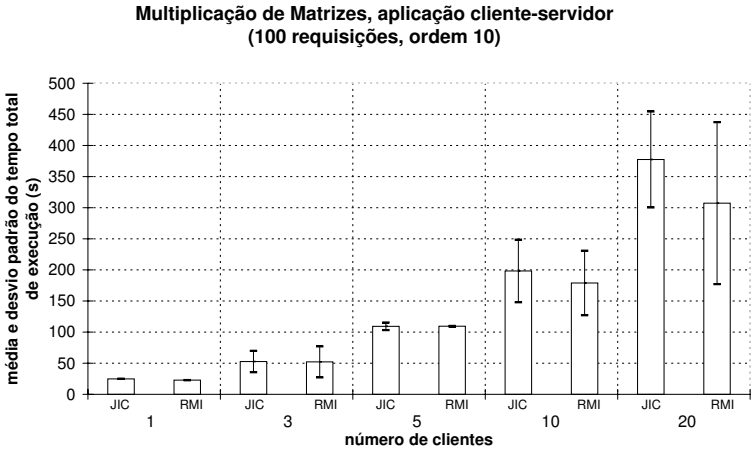
5.1.2 Métricas

Em ambos os cenários, as métricas analisadas consistiram na *média* e *desvio padrão* dos *tempos totais de processamento dos clientes/peers*. Através da média, é possível verificar se, com o JIC, o comportamento do sistema melhora no que diz respeito a tempo total de execução de todas as requisições. Caso isto aconteça, temos que os clientes/*peers* conseguem, em média, executar todo o conjunto de requisições em menos tempo. O desvio padrão representa a dispersão dos tempos de execução em torno da média. Valores altos para o desvio padrão significam que existe uma grande dispersão dos tempos individuais para cada cliente/*peer*, de forma que um usuário pode perceber diferentes responsividades para uma mesma carga de trabalho que é executada pelo sistema distribuído.

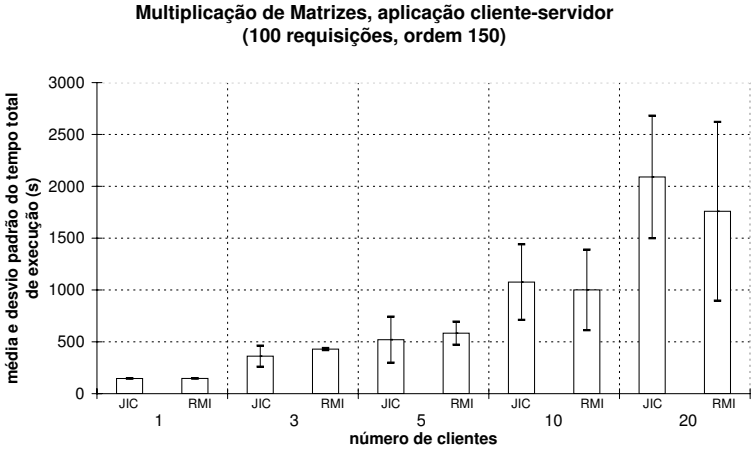
Portanto, cada cliente/*peer* registra o tempo entre a elaboração da primeira requisição e o término da última, contabilizando o tempo total gasto para a execução de todas as 100 requisições, ou seja, o tempo total de execução da aplicação. Baseando-se nestes tempos individuais e considerando-se a quantidade de clientes/*peers* de cada cenário, calculamos a média e o desvio padrão para os dados coletados.

5.1.3 Resultados

A Figura 5.1(a) mostra os resultados dos experimentos realizados sobre a aplicação cliente-servidor para um tamanho pequeno de matrizes. O eixo x representa a quantidade de clientes executando requisições simultâneas no servidor. O eixo y representa a média e desvio padrão dos tempos de processamento considerando todos os clientes, assim como descrito na seção 5.1.2. A média dos tempos de execução está representada através do gráfico de barras e o desvio padrão através de uma barra de erros. Ambos os valores estão representados em segundos.



(a)



(b)

Figura 5.1: (a) Aplicação cliente-servidor para matriz pequena (ordem 10) e (b) para matriz grande (ordem 150)

Como pode ser visto, a aplicação cliente servidor implementada através de JIC tem desempenho comparável a RMI. Os tempos um pouco maiores são justificados pela existência do servidor Jabber, o qual age como um *relay* para todas as mensagens trocadas entre entidades JIC. Entretanto, consideramos que estamos diante de um *trade-off* vantajoso, uma vez que um ponto de *relay* faz com que a solução JIC lide bem com *firewalls* e NATs, como mencionado no Capítulo 4. Além disso, ao desenvolver o código da aplicação que usa JIC, não foi necessário lidar com quaisquer preocupações referentes a programação concorrente, pois optamos por criar *Access Points* com uma única thread para executar todas as requisições. No caso da aplicação construída sobre RMI, foi preciso lidar com programação concorrente, uma vez que o servidor cria uma nova thread para cada nova requisição. Como as requisições feitas pelos clientes são bloqueantes, quando n clientes estão fazendo requisições simultâneas, temos n threads executando no lado servidor. Mesmo em aplicações simples, é preciso pelo menos pensar nas situações em que há a possibilidade de condições de corrida. Na medida em que as aplicações ficam mais complexas, mais difícil se torna o controle de suas regiões críticas.

É importante notar que o desvio padrão observado na execução do cenário com o JIC se deve ao fato de que as requisições são colocadas em uma única fila para serem processadas sequencialmente. Cada cliente, ao iniciar, dispara todas as suas requisições, imediatamente, de forma que o tempo total de execução da aplicação depende do tempo em fila da última requisição. Logo, clientes que ficam com requisições no final da fila, demoram mais para finalizar a execução da aplicação. No caso da aplicação implementada sobre RMI, cada cliente dispara uma requisição, bloqueia até o recebimento do resultado e só depois dispara a próxima requisição. Diante disso, para RMI, esperávamos que os tempos de cada requisição fossem bastante próximos, com um desvio padrão mínimo. Porém, este não foi o resultado obtido. Observamos que, quanto maior o número de clientes, maior o desvio padrão do tempo de execução de uma requisição, resultando também em um maior desvio padrão nos tempos totais de execução da aplicação. Acreditamos que tal variação se deve ao fato de que o controle de exclusão mútua é realizado por blocos *synchronized* de Java, de forma que a escolha da próxima thread a adquirir o *lock* é aleatória. Sendo assim, algumas requisições demoram muito tempo para executar porque não conseguem adquirir o *lock* rapidamente.

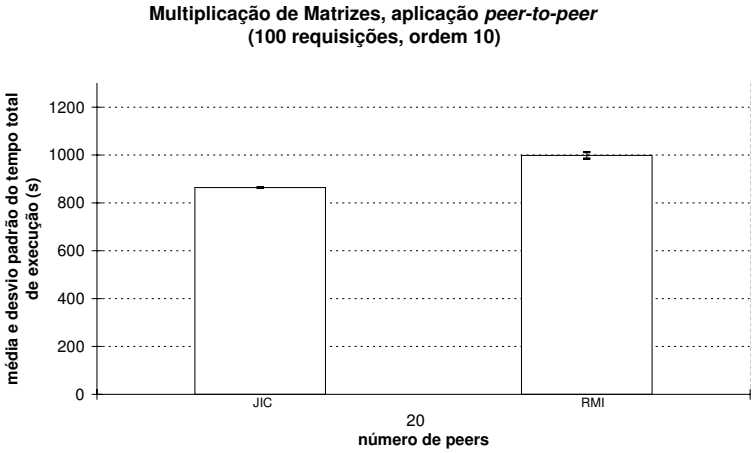
Como podemos ver na Figura 5.1(b), o mesmo resultado pode ser verificado para os

experimentos que usam um tamanho maior de matriz.

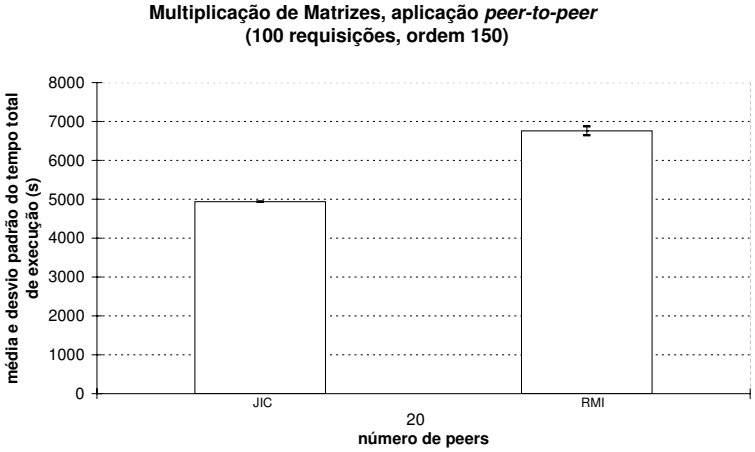
Os resultados obtidos através do experimento com aplicações *peer-to-peer* são mostrados nas Figuras 5.2(a) e 5.2(b). O cenário deste experimento era composto por uma rede de 20 *peers*, conectados em clique. Em ambas as granularidades de requisições podemos observar que o tempo total de execução da aplicação (100 requisições) é menor na aplicação construída sobre o JIC. Creditamos este resultado à natureza síncrona da aplicação construída sobre RMI.

Na execução da aplicação JIC, cada *peer*, ao receber uma requisição, a processa e rapidamente a encaminha para outros três *peers* aleatórios. As requisições são colocadas nas filas dos *peers* que as receberam e o *peer* que as disparou desbloqueia imediatamente. Neste momento, a thread que realizou estes passos volta a consumir um evento da fila do *Access Point* e processa uma nova requisição. Note que a rede *peer-to-peer* é formada por 20 *peers*, cada um com uma única thread, num total de 20 threads simultâneas processando uma requisição por vez.

Na execução da aplicação Java RMI, cada *peer*, ao receber uma requisição, também a processa e a encaminha para outros três *peers* aleatórios. Porém, este encaminhamento é feito através de uma chamada bloqueante, visto que Java RMI realiza chamadas síncronas. Isto significa que o encaminhamento da requisição para o segundo *peer* aleatório só é feito quando a chamada realizada no primeiro *peer* retorna. No entanto, a thread que disparou a requisição para o primeiro *peer* está bloqueada e só irá desbloquear quando o *peer* contactado retornar o resultado de sua requisição. Mas o *peer* contactado também irá processar a requisição e contactar outros três *peers*, bloqueando ao disparar a requisição para o primeiro e dando início ao mesmo procedimento. Note que, em toda a rede, não existe mais de um *peer* executando uma mesma requisição em um mesmo momento, ou seja, cada requisição é executada na rede de maneira sequencial. Logo, para uma rede de n *peers*, existem no máximo n^2 threads, sendo que no máximo n estão executando alguma requisição. As demais estão bloqueadas à espera da resposta de um outro *peer* na rede. Para uma rede *peer-to-peer* formada por 20 *peers*, temos cada um com no máximo 20 threads, num total de 400 threads simultâneas, sendo que no máximo 20 estão processando uma requisição. A quantidade de threads simultâneas executando uma requisição é a mesma observada no experimento com o JIC. Porém, os *peers* construídos sobre Java RMI precisam lidar com o *overhead* de criação



(a)



(b)

Figura 5.2: (a) Aplicação *peer-to-peer* para matriz pequena (ordem 10) e (b) para matriz grande (ordem 150)

e gerenciamento de threads, além das trocas de contexto e sincronização entre as mesmas, *overhead* este que não é observado na aplicação construída sobre JIC.

5.2 Avaliação de Engenharia de Software

A avaliação que considera métricas de engenharia de software foi realizada sobre o OurGrid. Em suas primeiras versões, a comunicação entre entidades remotas era realizada através de Java RMI, ou seja, a interação entre objetos remotos acontecia através de chamadas bloqueantes. No entanto, este modelo de comunicação não é o mais apropriado para o OurGrid, uma vez que a maioria dos seus componentes poderia prosseguir com algum processamento local paralelamente às invocações remotas em andamento. Nestas versões, paralelismo era explorado através de múltiplos threads.

Uma alternativa pensada consistia em realizar chamadas de métodos remotos sem retorno (métodos *void*), de maneira que respostas às requisições eram entregues através de *callbacks*. Contudo, métodos remotos sem retorno também fazem com que o cliente bloqueie até que o código remoto seja completamente processado. Isso significa que o processamento remoto decorrente da invocação de um método deve ser desacoplado de tal invocação. Uma boa solução para esta necessidade consiste em fazer a execução do método remoto apenas criar uma requisição e colocá-la em uma fila para ser processada posteriormente, fazendo com que o cliente desbloqueie em seguida. Diante desta idéia, um padrão de programação distribuída baseado na arquitetura *event-driven* foi implementado a partir da versão 2.2 do OurGrid. Através deste padrão, é possível simular comunicação assíncrona sobre uma solução de objetos distribuídos, de forma que as *threads* passam a ter o escopo bem definido, não ultrapassando as fronteiras do módulo ao qual pertencem. A Figura 5.3 apresenta o padrão *event-driven* usado no OurGrid 2.2 para o componente `MyGrid`:

Através desta arquitetura, cada componente remoto possui um objeto que implementa uma interface remota (Java RMI), exportando sua funcionalidade para ser acessada a partir de outros processos. No caso da Figura 5.3, o objeto remoto `MyGridImpl` implementa a interface remota `MyGrid`. Ao receber uma invocação remota, o objeto `MyGridImpl` repassa esta invocação para o módulo `MyGrid` através de uma *façade*, no caso, a `MyGridFaçade`. Note que a `MyGridFaçade` precisa implementar a mesma interface que o objeto remoto,

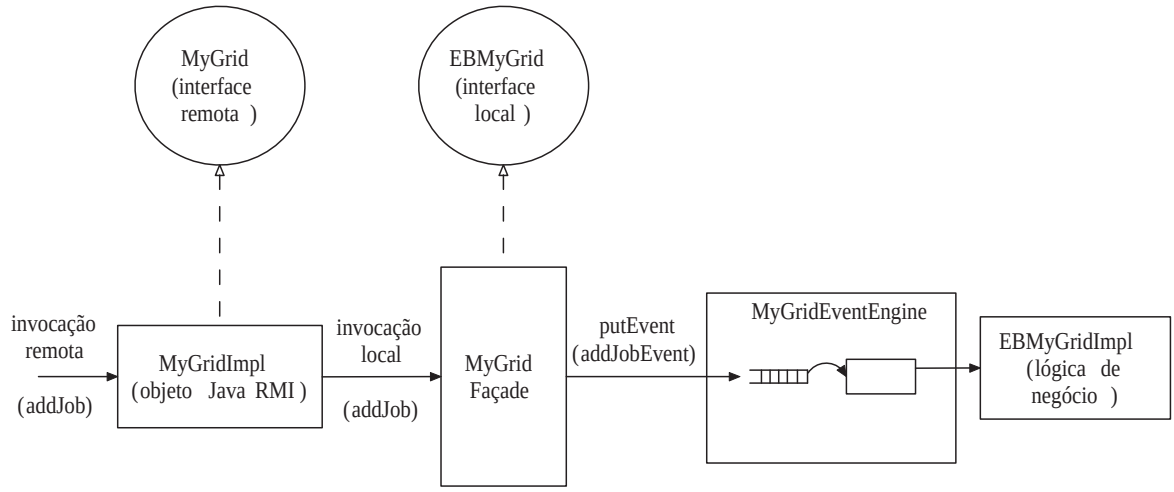


Figura 5.3: Padrão *event-driven* usado no OurGrid 2.2.

para que os métodos remotos possam ser delegadas para a mesma. No entanto, *MyGrid-Façade* é um objeto local, que não precisa implementar uma interface remota, nem lançar `RemoteExceptions` em seus métodos. Por isso, para cada interface remota, existe uma interface *event-based* equivalente (neste caso, `EBMyGrid`). A *façade* cria um evento que representa o método invocado e o coloca na fila de eventos do `EventEngine`, retornando e desbloqueando a thread que o Java RMI criou para esta invocação. O `EventEngine` possui uma ou mais threads internas (em nossa implementação, todos os módulos possuem apenas uma), que consomem os eventos da fila e invocam o objeto alvo, que implementa a lógica de negócios (`EBMyGridImpl`). Ao terminar a execução do método, uma thread do `EventEngine` consome outro evento da fila, ou bloqueia na mesma, caso não haja eventos.

Todavia, com uso deste padrão, o mecanismo detecção de falhas embutido nas chamadas bloqueantes deixa de ser suficiente para monitorar os objetos de interesse. Portanto, é preciso usar um mecanismo que monitore objetos remotos de maneira desacoplada à invocação. Considerando esta necessidade, a partir da versão 2.2 do OurGrid, um mecanismo de detecção de falhas foi desenvolvido. A partir da versão 3.3, este mecanismo foi evoluído, permitindo o uso de modelos mais flexíveis de detecção de falhas, como os mencionados na seção 2.4.1. Juntamente com o padrão *event-driven* até então usado, este novo mecanismo foi encapsulado dentro do *middleware* de comunicação a partir da versão 4.0, com o uso do JIC. Esta abordagem evita que o programador tenha que desenvolver manualmente o código que implementa o padrão e o próprio mecanismo de detecção de falhas.

Grosso modo, analisamos o software em quatro versões: (i) a versão 2.1.3, que usa RMI puramente, (ii) a versão 2.2, que possui aproximadamente o mesmo conjunto de funcionalidades da versão 2.1.3 e implementa o padrão *event-driven* sobre RMI, (iii) a versão 3.3, que é uma evolução da 2.2 por possuir um conjunto de funcionalidades diferente e algumas correções de *bugs* e (iv) a versão 4.0, que usa JIC como *middleware* de comunicação e possui aproximadamente o mesmo conjunto de funcionalidades da versão 3.3.

Note que, apesar de analisarmos quatro versões, do ponto de vista de engenharia de software temos três momentos distintos que representam mudanças arquiteturais importantes: (i) o uso de RMI puro (versão 2.1.3), (ii) o uso de RMI com o padrão *event-driven* (versões 2.2 e 3.3) e (iii) o uso de JIC (versão 4.0). Isso significa que compararemos a versão 2.1.3 com a 2.2 e a versão 3.3 com a 4.0. As versões 2.2 e 3.3 possuem uma mesma arquitetura, porém há uma grande mudança no conjunto de funcionalidades entre elas. Em ambas, a arquitetura *event-driven* é usada. Mas nem todos os componentes implementam esta arquitetura devido à exigência de portas abertas no *firewall* para retorno via *callback*. A Tabela 5.1 sumariza as características das versões analisadas:

Versão	Comunicação			Detecção de falhas			Modificação de
(OurGrid)	RMI	RMI + padrão	JIC	RMI	desacoplada	JIC	Funcionalidades
2.1.3	x	-	-	x	-	-	-
2.2	-	x	-	-	x	-	não
3.3	-	x	-	-	x	-	sim
4.0	-	-	x	-	-	x	não

Tabela 5.1: Características das versões do OurGrid analisadas. A última coluna indica se houve mudanças de funcionalidades em relação à versão anterior analisada. A versão 2.1.3 contém o conjunto inicial de funcionalidades. Detalhe: na tabela, RMI representa "Java RMI". Da mesma forma, "RMI + padrão" significa "Java RMI + padrão *event-driven*".

5.2.1 Métricas

Ao realizar uma análise de engenharia de software para comparar uma solução que usa JIC com uma solução que implementa objetos distribuídos, mais precisamente Java RMI, quere-

mos responder as seguintes perguntas: *usando JIC, é mais fácil programar uma aplicação distribuída complexa?* Além disso, *usando JIC, o programador concentra-se mais na lógica de negócio da aplicação, precisando lidar menos com funcionalidades transversais às funcionalidades da mesma?*

Durante a avaliação, coletamos algumas métricas que nos ajudarão a responder estas perguntas, incluindo algumas métricas básicas, que nos ajudam a ter a idéia da variação do tamanho do código tanto em linhas de código, como em número de classes. Eis as métricas:

Número de linhas de código (LOC, *Lines Of Code*) : Representa o número total de linhas de código da aplicação que está sendo analisada. Ao comparar duas versões de software através desta métrica, é preciso verificar se ambas possuem a mesma formatação em relação ao tamanho máximo de uma linha. De maneira análoga, também é preciso verificar se os estilos de codificação das versões analisadas são os mesmos (por exemplo, se as chaves iniciais de um bloco ocupam uma nova linha). Diferenças de estilo e formatação podem inserir erros que comprometem os resultados da análise. Nesta avaliação, ao contabilizar o número de linhas, não são consideradas linhas em branco nem comentários.

Número de classes : Uma vez que estamos avaliando soluções construídas sobre uma linguagem orientada a objetos, esta métrica representa o número de *classes de objetos* codificadas pelo programador. Ao realizar esta avaliação, tanto classes quanto interfaces Java são consideradas. Classes geradas automaticamente em tempo de execução ou através de um compilador (e.g., RMIC) não fazem parte desta medição. Mudanças no número de classes e/ou no número de linhas de código na avaliação de versões com um mesmo conjunto de funcionalidades podem refletir a influência das mudanças arquiteturais e/ou de *design* entre tais versões.

Número de blocos *synchronized* : Representa o número de vezes que o programador precisou usar um mecanismo de exclusão mútua para controlar o acesso a regiões críticas do código, evitando que recursos não compartilháveis sejam acessados simultaneamente por mais de uma *thread*. Para efeito de contagem, métodos *synchronized* são contabilizados como blocos *synchronized*. A cada bloco usado, o programador está desviando sua atenção da lógica de negócio para lidar com detalhes referentes a pro-

gramação concorrente, precisando raciocinar sobre detalhes tais como quais *threads* têm acesso a que trechos de código, quais trechos de código podem modificar estado compartilhado por mais de uma *thread*, quais *locks* de objetos já foram adquiridos por uma *thread* e em que ordem os *locks* foram adquiridos antes da *thread* entrar numa nova região crítica.

Complexidade das classes : Também conhecida como *weighted methods per class*, esta métrica indica a soma das complexidades dos métodos definidos em uma classe, representando a complexidade total da mesma. Classes com valores altos para esta métrica necessitaram de grande esforço para desenvolvimento, também requerendo grande esforço para manutenção. A medida de complexidade de cada método é feita através da métrica *Complexidade Ciclomática de McCabe* [Tho76], usada para indicar a *complexidade psicológica* [Bar05] de um método.

Complexidade Ciclomática de McCabe: Mede o número de caminhos lineares independentes dentro de um programa ou dentro de um módulo de um programa. O resultado desta medição é um número ordinal que pode ser comparado com a complexidade de outros programas. Em geral, o software a ser analisado através desta métrica é representado por um grafo em que cada vértice corresponde a uma expressão (ou comando) e cada aresta representa a transferência de controle entre vértices. Ramificações saindo de um vértice representam expressões condicionais. Ciclos no grafo significam que o número de caminhos pode ser infinito. Por isso, o cálculo da complexidade apenas considera "caminhos base", de forma que ciclos em um dado caminho são contados apenas uma vez. Sendo assim, a Complexidade Ciclomática ($v(G)$) pode ser calculada pela seguinte fórmula:

$$v(G) = P + 1 \tag{5.1}$$

onde:

P = número de expressões de decisão.

Alternativamente, pode-se usar uma outra fórmula:

$$v(G) = A - V + 2 \tag{5.2}$$

onde :

A = número de arestas do grafo; e

V = número de vértices do grafo.

Em geral, considera-se o valor 10 como o limite recomendado para a complexidade ciclomática de um método. Complexidades acima desse valor indicam que o método é muito difícil de ser mantido e testado [Tho76].

Separação de interesses : Dizemos que um software usa bem o conceito de separação de interesses (*separation of concerns*, princípio introduzido por Dijkstra [Eds76] e Parnas [Dav72]) quando está dividido em partes de maneira que cada parte tem suas responsabilidades bem definidas, sem que uma determinada parte seja detentora de várias responsabilidades ou uma responsabilidade esteja espalhada por várias partes. Para mensurar este conceito, usamos as duas métricas que seguem abaixo:

Espalhamento do código de tratamento de falhas: Quando uma funcionalidade é implementada de maneira espalhada por diversos trechos de código, dizemos que existe um problema conhecido como *code scattering*, ou seja, espalhamento de código [HL95]. Neste caso, o código que implementa tal funcionalidade (interesse) não está modularizado, estando espalhado pelo sistema. Este problema é comum no tratamento de falhas de componentes em sistemas implementados sobre objetos distribuídos, uma vez que a detecção de falhas está acoplada à chamada e a notificação é feita através do lançamento de exceções. Isso significa que o tratamento de falhas está espalhado pela aplicação em cada bloco `try-catch` que encapsula uma invocação remota. No caso de aplicações construídas sobre objetos distribuídos, cada `try-catch` que encapsula o tratamento de uma exceção de comunicação representa um trecho de tratamento de falhas. Logo, para cada entidade passível de falha (entidade remota), podemos medir (i) quantos trechos de código tratam a falha desta entidade, (ii) quantas classes diferentes realizam este tratamento e (iii) o número médio de trechos de código `try-catch` por classe que realizam este tratamento.

Desvio da lógica de negócio: Conceito conhecido como *obliviousness*, define quão ciente o código está de preocupações transversais a funcionalidades, ou seja, o quanto o código da aplicação conhece ou lida com aspectos que não são inerentes

às funcionalidades "alvo". Analisamos esta métrica de acordo com a quantidade de código desenvolvido pelo programador da aplicação que não lida com funcionalidades inerentes à mesma. Logo, analisamos (i) a porção de código (número de linhas de código e número de classes) que implementa alguma preocupação transversal à funcionalidade e (ii) o número de referências a esta implementação (referências Java) transversal. Quanto menos código implementar funcionalidades transversais e quanto menos referências houver para tal código, menos o programador se desvia da lógica de negócio. No contexto do OurGrid, mensuramos desvios decorrentes da implementação do padrão *event-driven* e do mecanismo de detecção de falhas.

Com exceção da métrica "número de blocos *synchronized*", todas as métricas foram coletadas com ajuda do *Eclipse Metrics Plugin*, versão 1.3.6 [Fra06]. As medidas de separação de interesse foram coletadas através do mecanismo de busca do Eclipse [Ecl06], mas com ajuda do *Eclipse Metrics Plugin* para realizar contagem de linhas de código.

5.2.2 Resultados

Assim como mencionado na seção 5.2.1, a primeira pergunta que queremos responder ao comparar uma solução implementada sobre JIC com outra implementada sobre Java RMI é: *usando JIC, é mais fácil programar uma aplicação distribuída complexa?* Diante da experiência com o JIC e RMI, acreditamos que programar usando JIC é mais fácil porquê:

(I) O programador não precisa lidar com os problemas inerentes ao modelo de threads, como mencionado no Capítulo 1. É possível desenvolver uma aplicação inteira sem se preocupar com quaisquer detalhes relacionados à programação concorrente, ou seja, sem escrever código multi-threaded. Isto é alcançado de maneira bastante simples, pois é apenas necessário que o o programador configure cada *Access Point* da aplicação para que tenha somente uma thread. Neste caso, mesmo sem código multi-threaded, uma aplicação poderá ter todo o seu paralelismo explorado, desde que tenha uma quantidade suficiente de *Access Points*. Se tal paralelismo não estiver sendo totalmente explorado, o programador pode perfilar a aplicação e apenas investir esforços em código multi-threaded para os *Access Points* em que há gargalos. Para avaliar este item, iremos usar como métrica o *número de blocos synchronized*.

(II) Com JIC, o tratamento de falhas para uma entidade remota realizado em um cliente encontra-se em um único trecho de código. Vale salientar que, uma vez que as soluções baseadas em objetos distribuídos usam detecção de falhas acoplada à invocação, cada invocação remota realizada por um cliente deverá tratar uma possível exceção que indica a falha da entidade com a qual o cliente está se comunicando. No caso de Java RMI, cada invocação remota deve ser tratada através de um bloco `try-catch`, ou o método que a executa deve passar a `RemoteException` adiante. Em geral, o uso de `try-catchs` faz com que o tratamento de falhas fique espalhado pelo código, tornando-o menos legível, além de dificultar seu entendimento e manutenção. Lidar com trechos de código que fazem operações semelhantes, mas de maneira espalhada facilita a inserção de *bugs* em softwares. Através do JIC, o tratamento de falhas de entidades é centralizado em um único método por classe. Ao codificar este método, o programador deve raciocinar sobre os possíveis estados da aplicação no momento da falha e tomar a decisão adequada. Note que este esforço é necessário de qualquer forma, por se tratar de um ambiente distribuído, e força o programador a raciocinar sobre o comportamento da aplicação diante dos possíveis estímulos. Além disso, existe um outro problema das soluções baseadas em objetos distribuídos relacionado à concorrência e detecção de falhas. Ao receber uma exceção que indica a falha de um componente, o cliente deverá gerenciar seu estado. Contudo, se mais de uma thread recebe uma exceção ao se comunicar com o mesmo objeto remoto, ambas irão gerenciar o seu estado para voltar a um estado consistente. Porém, se duas threads estão tentando modificar um mesmo estado, é possível que se tenha condições de corrida. O programador, portanto, precisa analisar a existência de regiões críticas e tratá-las, se existirem. O JIC não herda tais problemas caso seus *Access Points* tenham apenas uma thread, pois os eventos de notificação são enfileirados para serem consumidos sequencialmente e assim a notificação acontece uma vez por entidade. Para avaliar este item, será usada a métrica *espalhamento do código de tratamento de falhas*.

(III) Em aplicações assíncronas usando JIC, não é preciso implementar um mecanismo próprio de detecção de falhas, nem recorrer a mecanismos de terceiros, assim como é preciso ao usar o padrão *event-driven* com RMI. Implementar um mecanismo próprio requer tempo e esforço, além de desviar o programador da lógica de negócio. Para

analisar este item, iremos usar a métrica *desvio da lógica de negócios*.

(IV) Não existe mudança significativa nas complexidades das classes e dos métodos de uma aplicação que usa JIC em relação a uma aplicação que é implementada sobre objetos distribuídos. Apesar de o programador precisar gerenciar sua aplicação como uma máquina de estados, acreditamos que esta prática não torna o JIC um software mais complexo. Pelo contrário, acreditamos que esta prática leva o programador a ter uma boa disciplina ao realizar programação distribuída. Para verificar este item, usaremos a métrica *complexidade das classes*.

Na intenção de responder a segunda questão (*se, usando JIC, o programador concentra-se mais na lógica de negócio da aplicação, precisando lidar menos com funcionalidades transversais às funcionalidades da mesma*), acreditamos que o programador que usa JIC se concentra mais na lógica de negócio da aplicação do que o programador que usa soluções baseadas em objetos distribuídos, pois este último se concentra muito em funcionalidades transversais às funcionalidades da mesma. Basicamente, o principal argumento que nos motiva a responder esta questão positivamente é:

(V) Acreditamos que o código de uma aplicação distribuída assíncrona desenvolvida sobre JIC está menos ciente de preocupações transversais a funcionalidades do que um código desenvolvido sobre soluções baseadas em objetos distribuídos. Caso isto seja verdade, significa que o código da aplicação conhece pouco ou lida pouco com aspectos que não representam funcionalidades da mesma. Para avaliar este último item, iremos analisar a métrica *desvio da lógica de negócio*.

Juntamente com os itens (I), (II) e (III) mencionados acima, o item (V) nos ajudará a responder este segundo questionamento.

OurGrid-2.1.3 X OurGrid-2.2

A avaliação entre as versões 2.1.3 e 2.2 do OurGrid mostram resultados intermediários no processo de adoção do JIC. Ao comparar tais versões, estamos verificando o impacto causado pela troca de uma solução puramente baseada em objetos distribuídos por um modelo intermediário, que ainda se comunica através de objetos distribuídos, mas que implementa

uma arquitetura *event-driven*, codificada pelo programador da aplicação. O próximo passo (evolução para o JIC) consiste em eliminar o uso de objetos distribuídos e usar a arquitetura *event-driven* oferecida pela camada de comunicação, que fornece toda a infra-estrutura necessária (e.g., detecção de falhas).

A primeira verificação realizada diz respeito à quantidade de linhas de código e o número de classes. Queremos verificar o impacto das mudanças arquiteturais no tamanho do código. A Tabela 5.2 apresenta os primeiros resultados:

	OurGrid 2.1.3	OurGrid 2.2
Número de linhas de código	33156	34734
Número de classes	302	501

Tabela 5.2: Comparação do tamanho do código entre as versões 2.1.3 e 2.2 do OurGrid.

Note que a versão 2.2 possui um aumento pouco significativo na quantidade de linhas de código (4,75%). No entanto, a quantidade de classes aumentou 65,9%, ou seja, em 199 classes. Podemos atribuir tal aumento à codificação do padrão *event-driven* sobre Java RMI. Considerando as principais entidades que constituem o padrão (eventos, *façades*, *event engines*, filas e classes *event-based*), temos um aumento de 88 classes. Deste total, 56 são classes de evento, as quais possuem poucas linhas de código (12,45 linhas por classe em média). Isso explica porque o aumento do número de linhas de código não acompanhou o aumento do número de classes.

Uma vez que os estímulos aplicáveis a cada entidade foram definidos através de eventos, o programador pôde modularizar melhor a aplicação. Foram definidos módulos explícitos e cada entidade passou a ter suas responsabilidades mais bem definidas. Tal divisão de responsabilidades implicou um aumento no número de classes, complementando o aumento observado pelo próprio padrão. Logo, a implantação do padrão *event-driven* resultou em mais disciplina para o programador. Contudo, como veremos adiante, o preço dos benefícios conseguidos com o padrão é a necessidade de desenvolver manualmente código que não está relacionado à lógica de negócio da aplicação.

Apesar de ser necessário lidar com questões transversais às funcionalidades, o uso do padrão *event-driven* permite que o programador lide menos com os problemas relacionados

à concorrência. A princípio, temos que os únicos pontos de sincronização são as filas de evento de cada módulo. Considerando que cada módulo possui apenas uma thread e os módulos se comunicam apenas através da *façade*, o programador não precisa lidar com quaisquer mecanismos para fazer sua aplicação *thread-safe*. Além disso, fica mais fácil raciocinar sobre o comportamento da aplicação, uma vez que as fronteiras das threads estão delimitadas pelos módulos. Isso significa que, mesmo que o programador crie módulos multi-threaded, o escopo que deverá considerar ao lidar com concorrência será o próprio módulo. Juntamente com estes benefícios, verificamos também que o número de blocos *synchronized* diminuiu consideravelmente, assim como mostra a Tabela 5.3:

	OurGrid 2.1.3	OurGrid 2.2
Número de blocos *synchronized*	174	110

Tabela 5.3: Número de blocos synchronized nas versões 2.1.3 e 2.2 do OurGrid.

A diminuição em 36,78% mostra que o programador precisa lidar menos com concorrência. Vale salientar que boa parte dos blocos `synchronized` usados na versão 2.2 não requerem que o programador empregue grande esforço raciocinando sobre threads. Em sua grande maioria, são aplicados nas filas de evento seguindo a lógica previamente definida pelo padrão *event-driven*.

Em relação ao espalhamento do código de tratamento de falhas, a versão 2.1.3 usa apenas o mecanismo embutido na invocação de Java RMI. A versão 2.2, por sua vez, possui um mecanismo próprio que permite realizar detecção de falhas. Contudo, uma vez que a versão 2.2 foi construída sobre Java RMI, à cada invocação, ainda é preciso tratar exceções remotas através de blocos `try-catch`. Isso significa que, mesmo usando um mecanismo próprio para detecção de falhas, a versão 2.2 ainda precisa lidar com o tratamento de falhas via Java RMI, caso aconteça uma falha exatamente no momento de uma invocação remota.

A detecção de falhas através do mecanismo da versão 2.2 é baseada na invocação de métodos remotos de Java RMI. As entidades que devem ser monitoradas implementam uma interface que contém um método `ping`, cuja implementação é vazia. Uma invocação remota bem sucedida a este método significa que a entidade invocada está no ar. Caso esta invocação resulte em uma `RemoteException`, a falha é detectada, de maneira acoplada à invocação.

As Tabelas 5.4 e 5.5 mostram as medições em relação ao espalhamento do código de detecção de falhas para as versões 2.1.3 e 2.2 do OurGrid, sem considerar ainda o mecanismo próprio da versão 2.2. Note que o OurGrid é dividido em três entidades principais, cada qual executando um ou mais objetos remotos em uma JVM (*Java Virtual Machine*) própria.

OurGrid 2.1.3	MyGrid	Peer	GuM
Blocos try-catch	31	18	38
Classes	23	12	10
Média (blocos/classe)	1,35	1,5	3,8
Máximo (blocos/classe)	2	4	16

Tabela 5.4: Espalhamento do código de detecção de falhas para a versão 2.1.3 do OurGrid. Cada entidade (MyGrid, Peer e GuM) representa um conjunto de objetos remotos em JVMs diferentes.

OurGrid 2.2	MyGrid	Peer	GuM
Blocos try-catch	39	7	35
Classes	24	6	9
Média (blocos/classe)	1,625	1,17	3,89
Máximo (blocos/classe)	5	2	17

Tabela 5.5: Espalhamento do código de detecção de falhas para a versão 2.2 do OurGrid. Cada entidade (MyGrid, Peer e GuM) representa um conjunto de objetos remotos em JVMs diferentes.

Note que a média (blocos `try-catch` por classe) sofreu variações mínimas em todos os componentes. Todavia, é importante salientar que para ambas as versões, existem classes que possuem grandes quantidades de trechos de código para tratar falhas de uma mesma entidade. Esperamos que com o uso do JIC, este espalhamento seja minimizado, de forma que existirá no máximo um trecho de código por classe.

Se adicionarmos a esta análise o mecanismo de detecção de falhas próprio da versão 2.2, veremos que tal mecanismo não minimiza o problema de espalhamento de código observado acima. Verificando o código da versão 2.2, foi percebemos que este mecanismo possui 4

classes que definem métodos `ping`. Ao todo, existem 27 trechos de código que invocam estes métodos e realizam o tratamento em caso de falha. Todos os trechos referenciados pertencem ao componente GuM, do OurGrid. Logo, temos em média, 2,25 trechos de código por classe, os quais tratam da falha do componente GuM. Note, ao comparar com as Tabelas 5.4 e 5.5, que o nível de espalhamento é semelhante aos níveis observados no mecanismo padrão de Java RMI.

A próxima métrica analisada tem o objetivo de identificar a quantidade de código da aplicação que não tem relação com a lógica de negócio. Para cada funcionalidade transversal à lógica de negócio da aplicação, mediremos o número de classes que implementam esta funcionalidade, bem como o número de linhas de código e a quantidade de referências Java para tal funcionalidade. Consideramos que duas preocupações básicas desviam o programador da lógica de negócio do OurGrid: (i) implementação do modelo *event-driven* e (ii) a implementação de um mecanismo de detecção de falhas próprio para substituir o mecanismo padrão do Java RMI.

Considerando que não existe padrão *event-driven* na versão 2.1.3, e que a detecção de falhas é realizada através dos próprios *timeouts* de Java RMI, não estamos considerando desvio da lógica de negócio para esta versão. Todavia, esta solução faz uso de um modelo de comunicação bloqueante (que não é adequado para aplicação), não possui um mecanismo de detecção de falhas flexível e configurável, e possui todos os problemas relacionados a threads no Capítulo 3. De todas as versões em análise, esta é, provavelmente, a que oferece mais complicações no que diz respeito à concorrência, desviando o programador da lógica de negócio. Portanto, é importante salientar que estamos realizando uma comparação usando um cenário favorável à versão 2.1.3, uma vez que não consideramos desvio de lógica decorrente de preocupação com código concorrente.

A versão 2.2, uma vez que implementa o modelo *event-driven*, possui outras variáveis a serem analisadas. A Tabela 5.6 representa a porção de código necessária para implementar o mecanismo de detecção de falhas e o padrão *event-driven*, detalhando cada componente necessário para codificação deste último.

Apesar dos benefícios trazidos pelo padrão event-driven, 20,61% do código da aplicação não diz respeito à lógica de negócio, o que representa mais de 7000 linhas de código. Em média, temos 10,42 referências no código para cada classe que não faz parte da lógica da

OurGrid 2.2	LOC	Classes	Referências	Média	Porcentagem da aplicação
Façades	1855	17	177	10,4	5,34%
Event-based	14	2	35	17,5	0,04%
EventEngines	775	11	42	3,81	2,23%
Fila de eventos	173	2	220	110	0,5%
Eventos	697	56	508	9,07	2,01%
Detecção de Falhas	3644	43	383	8,9	10,49%
Total	7158	131	1365	10,42	20,61%

Tabela 5.6: Porção da aplicação que lida com preocupações transversais às funcionalidades do OurGrid 2.2. Cada linha representa um componente do padrão *event-driven*

negócio. Este é o principal resultado que traduz a necessidade de se ter uma infra-estrutura de comunicação que encapsule o padrão *event-driven* juntamente com detecção de falhas.

A avaliação de complexidade das classes é feita através da métrica *Weighted Methods per Class* (WMC), que é calculada com base na complexidade ciclomática de McCabe. Considera-se que valores acima de 50 para uma classe indicam que a classe precisa ser refatorada, sendo dividida em duas ou mais classes. A Tabela 5.7 apresenta os resultados da análise de complexidade para as versões em avaliação.

WMC	OurGrid 2.1.3	OurGrid 2.2
Total (soma)	6004	5695
Valor médio por classe	21,68	13,34
Desvio padrão	31,27	15,03
Valor máximo por classe	249	82
Classes com WMC > 50	25	12

Tabela 5.7: Complexidade das classes na versão 2.1.3 e 2.2

Estes resultados reafirmam o fato de que o código da versão 2.2 está mais bem modularizado quando comparado à versão 2.1.3. A diminuição significativa da complexidade do código reflete a disciplina que o programador passa a ter quando usa o padrão *event-driven*. Apesar do maior número de classes na versão 2.2 (cada qual com suas responsabilidades

melhor definidas), a complexidade média por classe diminuiu em 38,47%. Apenas 12 classes excederam o valor limite, o que representa menos da metade do observado para a versão 2.1.3. Por fim, temos que o valor máximo observado na versão 2.1.3 é três vezes maior do que o valor observado para a versão 2.2. Tais valores significam que o código 2.2 requer menos tempo e esforço para manutenção, sendo um código mais fácil de entender e manter.

OurGrid 3.3 X OurGrid 4.0

A versão 4.0 do OurGrid representa a evolução para o uso do JIC. Em relação à versão 3.3, a versão 4.0 elimina o uso de objetos distribuídos como infra-estrutura de comunicação. Além disso, todo o código de detecção de falhas e implementação do modelo *event-driven* que antes era construído pelo programador, passou a ser encapsulado na infra-estrutura de comunicação, ou seja, no JIC. Note que os benefícios decorrentes do uso do JIC em aplicações que antes eram implementadas sobre objetos distribuídos incluem os benefícios obtidos com a versão 2.2 em relação à 2.1.3, além dos benefícios obtidos através da versão 4.0 em relação à 3.3 (esta última equivale arquiteturalmente à versão 2.2). Não foram feitas análises diretas entre as versões 2.1.3 e 4.0, devido ao conjunto distinto de funcionalidades.

É importante ressaltar que, em ambas as versões, alguns componentes (e.g., GUI e GuM) possuem threads extras internas a um módulo (na versão 3.3) ou *Access Point* (na versão 4.0). No caso da GUI, por exemplo, existe uma thread que recebe atualizações da aplicação e uma thread extra que responde rapidamente às ações do usuário. Já no caso da GuM, ao receber uma requisição, é criada uma nova thread para executá-la, de forma que a thread do módulo (ou *Access Point*) fica livre para receber operações especiais, como por exemplo, o aborto da execução corrente. Uma vez que decidiu-se usar componentes multi-threaded, parte do código ainda precisa de controle de exclusão mútua.

A avaliação sobre número de classes e número de linhas de código pode ser vista na Tabela 5.8:

Note que a diminuição na quantidade de linhas de código na versão 4.0 é bastante significativa. Comparado à versão 3.3, o OurGrid com JIC possui 20370 linhas de código a menos, o que representa uma diminuição de 41,79% de linhas de código. Em relação ao número de classes, a diminuição foi de 293 classes, o que equivale a uma diminuição de 40,36%.

	OurGrid 3.3	OurGrid 4.0
Número de linhas de código	48745	28375
Número de classes	726	433

Tabela 5.8: Comparação do tamanho do código entre as versões 3.3 e 4.0 do OurGrid.

Atribuímos esta diminuição ao código do padrão *event-driven* e do mecanismo de detecção de falhas que foram encapsulados dentro do JIC. Note também que, uma vez que o código da versão 3.3 já está bem modularizado (cada classe com sua responsabilidade), a diminuição do número de classes acompanhou a diminuição do número de linhas de código.

Em seguida, temos a medição do número de blocos *synchronized* entre estas duas versões, mostrada na Tabela 5.9:

	OurGrid 3.3	OurGrid 4.0
Número de blocos *synchronized*	179	66

Tabela 5.9: Número de blocos synchronized nas versões 3.3 e 4.0 do OurGrid.

Veja que a diminuição também é bastante significativa. O número de blocos *synchronized* entre as versões 3.3 e 4.0 diminuiu em 113 (63,13%). Entretanto, este número é muito alto, mesmo para a versão 3.3, que deveria usar blocos *synchronized* apenas na implementação do padrão *event-driven*. Observamos que este alto número se deve aos componentes que possuem threads extras. A GUI, por exemplo, possui 43 blocos `synchronized` na versão 3.3 e 44 blocos na versão 4.0. Note que isso equivale a 66% dos blocos existentes na versão 4.0.

No que diz respeito à detecção de falhas, temos que o tratamento realizado em ambas as versões é ligeiramente diferente. A versão 3.3 (assim como a 2.2), além de usar o mecanismo de detecção de falhas próprio, também realiza detecção de falhas através de blocos `try-catch`, uma vez que há comunicação através de Java RMI. A versão 4.0 usa apenas o mecanismo de detecção de falhas do JIC. Os mecanismos de detecção de falhas das versões 3.3 e 4.0 são semelhantes. Através deste, cada entidade precisa ter apenas um método por classe (método `notifyFailure`), que indica o tratamento a ser executado quando uma entidade falha. De maneira análoga, existe um tratamento para um serviço que aparece (ou

se recupera). Este tratamento também é adicionado através de um novo método por classe interessada (o método `notifyRecovery`). Note que, através de objetos distribuídos (sem mecanismos sofisticados para detecção de falhas), não existem mecanismos de notificação sobre a recuperação/aparecimento de serviços. O serviço, ao entrar no ar, deve anunciar seu aparecimento, ou as entidades devem tentar contactar o serviço de tempos em tempos para saber se está no ar. Portanto, a análise de espalhamento de código realizada sobre o código de detecção de falhas para os mecanismos próprios mostra que ambas as versões possui um espalhamento mínimo para todas as entidades. Sempre se tem a média de um método por classe para tratamento de falhas de entidades. A única consideração em relação a esta métrica diz respeito ao tratamento de `RemoteExceptions` na versão 3.3 decorrente do uso de Java RMI. Este espalhamento extra, imposto pela infra-estrutura de comunicação, é mostrado na A análise em relação ao espalhamento do código de detecção pode ser vista na Tabela 5.10.

OurGrid 3.3	MyGrid	Peer	GuM	Core Peer	Módulo DF
Blocos try-catch	29	41	48	6	13
Classes	9	18	16	6	9
Média (blocos/classe)	3,22	2,28	3,00	1,00	1,44

Tabela 5.10: Espalhamento do código de detecção de falhas para a versão 3.3 do OurGrid. Cada entidade representa um conjunto de objetos remotos em JVMs diferentes.

No que diz respeito ao desvio de código, consideramos que a versão 4.0, desenvolvida sobre o JIC, assim como a versão 2.1.3 não possui trechos de código que desviam da lógica de negócio. Em primeiro lugar, estamos desconsiderando os blocos *synchronized*, assim como fizemos com a versão 2.1.3 (vale ressaltar que a versão 4.0 possui um número muito inferior de blocos *synchronized*). Além disso, a versão 4.0 do OurGrid não possui mecanismo de detecção de falhas nem padrão *event-driven* codificados explicitamente. Como resultado, temos uma versão que permite comunicação não-bloqueante, possui um mecanismo de detecção de falhas flexível e configurável, permite comunicação na presença de *firewalls* e NATs, não possui os problemas de concorrência decorrentes do modelo de threads, e que não desvia o programador de sua lógica de negócios.

Todavia, ao usar a solução JIC, a maneira de raciocinar sobre a aplicação muda. A aplicação passa a ser vista como uma máquina de estados. Ao mesmo tempo em que acreditamos que este modo de pensar sobre a aplicação incentiva o programador a ter boa disciplina, não sabemos se esta prática aumenta a complexidade do código. Sendo assim, realizamos uma comparação de complexidade de código entre as versões 3.3 e 4.0, apresentadas na Tabela 5.11:

WMC	OurGrid 3.3	OurGrid 4.0
Total (soma)	7222	4046
Valor médio por classe	11,52	11,63
Desvio padrão	15,24	14,54
Valor máximo por classe	112	130
Classes com WMC > 50	22	9

Tabela 5.11: Complexidade das classes na versão 2.1.3 e 2.2

Tendo em vista que o número de classes da versão 3.3 é maior do que da 4.0, é natural que a soma das complexidades do programa seja mais alto na versão 3.3. Porém, se compararmos os outros valores, veremos que não houve mudança significativa de complexidades entre as versões 3.3 e 4.0. Isso significa que o uso do JIC não torna a aplicação mais complexa. No entanto, existe o argumento de que a versão 3.3 já foi projetada como uma máquina de estados, uma vez que já usa o padrão *event-driven*. Este argumento está correto, o que nos remete a comparar a versão 4.0 com a versão 2.1.3, que não foi codificada como máquina de estado. Se verificarmos os valores médios das complexidades das classes na versão 2.1.3, veremos que a versão 4.0 mantém níveis de complexidade ainda menores. O mesmo pode ser dito em relação ao desvio padrão da complexidade. Esta comparação, portanto, reforça a nossa impressão inicial de que programar uma aplicação como uma máquina de estados não incrementa a complexidade de um programa.

Capítulo 6

Conclusões e Trabalhos Futuros

Neste capítulo, apresentamos as considerações finais da dissertação, apresentando nossas impressões e direcionando possíveis trabalhos futuros, que complementem o trabalho apresentado ao longo deste documento.

6.1 Conclusões

Esta dissertação apresenta o JIC (*Java Internet Communication*), uma infra-estrutura de comunicação desenvolvida em Java que permite comunicação não-bloqueante através de um modelo de programação próximo ao paradigma orientado a objetos. A avaliação que realizamos sobre o uso do JIC suporta nossa tese de que *é possível combinar objetos distribuídos com a arquiteturas event-driven, resultando em uma infra-estrutura de comunicação assíncrona e bem integrada à linguagem de programação, na qual os problemas relacionados ao modelo de threads em ambientes distribuídos simplesmente não existem*.

Contudo, uma solução completa para sistemas distribuídos precisa endereçar outros aspectos inerentes ao ambiente em que executa. Em sistemas distribuídos, falhas parciais são possíveis, de forma que a infra-estrutura de comunicação deve fornecer um mecanismo robusto de detecção de falhas. Além disso, uma vez que a comunicação acontece entre vários domínios administrativos, o uso de *firewalls* e NATs passa a ser cada vez mais comum e segurança se torna um requisito essencial. Sendo assim, a infra-estrutura de comunicação deve oferecer mecanismos de segurança e permitir comunicação entre entidades remotas mesmo na presença de *firewalls* e NATs.

Diante disso, o JIC implementa um mecanismo de detecção de falhas flexível e configurável, oferecendo uma interface de alto nível para que a aplicação use este mecanismo. Segurança e comunicação na presença de *firewalls* e NATs são conseguidas através do uso do Jabber.

Aplicações desenvolvidas sobre JIC não possuem a mesma característica das aplicações tradicionais, nas quais existem fluxos de execução sequenciais bem definidos. Através do JIC, a aplicação é desenvolvida como uma máquina de estados, na qual o programador precisa definir de que maneira cada parte da aplicação reage a eventos externos. Acreditamos que definir os possíveis eventos e mapear o comportamento da aplicação baseando-se em tais eventos (e nos eventos de falha, que sempre existem), permite que o programador tenha uma boa disciplina de raciocinar sobre sua aplicação, não esquecendo de contemplar o comportamento dela mesmo nos casos menos comuns. Acreditamos que as soluções que buscam esconder o máximo de sistemas distribuídos do programador são "enganosamente" simples demais, fazendo com que o programador tenha que empregar muito esforço na correção de *bugs* que acontecem em casos que não foram pensados.

O JIC mostrou-se pronto para ser implantado no OurGrid, suprindo as características de uma infra-estrutura de comunicação necessárias a esta aplicação. Uma análise que considera alguns aspectos de Engenharia de Software nos mostrou que através do JIC, é possível construir um código mais modularizado e bem focado no negócio da aplicação, com a vantagem de não ser preciso lidar com problemas relacionados à concorrência. No que diz respeito a desempenho, o JIC mostrou-se comparável a Java RMI. A combinação destes dois resultados nos convenceu de que seu uso compensa.

6.2 Trabalhos Futuros

Identificamos pelo menos quatro aspectos relacionados ao JIC que podem ser explorados no futuro:

6.2.1 Evolução da implementação do JIC

Como todo bom software livre, o código do JIC deve permanecer em constante evolução. Atualmente, temos uma versão bem testada, usada em um ambiente interno de produção. No

entanto, alguns pequenos detalhes merecem a dedicação de um certo esforço, no intuito de melhorar o software. Nossa preocupação inicial sempre se voltou para construir um código que funcionasse. Desempenho não foi a preocupação inicial. Apesar disso, JIC mostrou que tem desempenho comparável a Java RMI, uma solução amplamente usada. Porém, temos conhecimento de alguns aspectos que podem melhorar desempenho, apesar de não termos priorizado nenhum até então.

O primeiro ponto consiste em evitar que a comunicação passe pelo servidor Jabber quando acontece entre duas entidades em um mesmo processo. Embora o servidor Jabber esteja no domínio local, existe um *overhead* associado à comunicação através do mesmo, além do fato de que mensagens do detector de falhas para objetos de um mesmo módulo são colocadas na fila do módulo, passando pelo servidor, caso não exista um mecanismo como o que sugerimos aqui.

Além disso, podemos evoluir o estabelecimento de conexão para que seja entre objetos. Atualmente, a conexão é estabelecida entre *Access Points*. Esta abordagem pode ser considerada por alguns como muito agressiva, pois a perda de uma mensagem entre dois *Access Points* faz com que todos os objetos nestes *Access Points* desconfiem de todos os objetos pertencentes ao outro *Access Point*.

Por fim, outra prática no sentido de evoluir o JIC consiste em perfilar alguns cenários de execução e identificar se existem pontos nos quais compensa o esforço de refatorar o código diante do possível ganho de desempenho.

6.2.2 Calibrar Mecanismo Atual de Detecção de Falhas

Na seção 4.5.6, apresentamos uma sugestão para construção de um experimento que execute no ambiente alvo de uma aplicação durante alguns dias. Esta sugestão serve como uma orientação ao usuário para que os parâmetros de detecção de falhas possam ser configurados de maneira mais adequada ao ambiente de execução.

Um possível trabalho futuro consiste em implementar o experimento e verificar se através do seu uso é possível manter a qualidade de detecção de falhas desejada. Um passo adiante pode consistir em implementar um experimento que roda de tempos em tempos juntamente com a aplicação no ambiente de produção e ajusta automaticamente os parâmetros de detecção de falhas, mantendo a qualidade de serviço esperada.

6.2.3 Estudar Efeitos do Mecanismo de Detecção Probabilístico

Atualmente, usamos um mecanismo de detecção alternativo ao mecanismo planejado inicialmente. Conforme mostramos na seção 4.5.6, encontramos dois problemas, um que diz respeito ao uso do mecanismo probabilístico enquanto a amostra não preencheu a janela e outro que diz respeito à sensibilidade do mecanismo em ambientes nos quais a distribuição da população aparentemente não é uma distribuição normal.

Consideramos que a investigação destes problemas e a proposta de alternativas representa uma pesquisa muito interessante e relevante.

6.2.4 Comunicação Síncrona na Periferia da Aplicação

A versão atual do JIC não fornece mecanismos que possibilitem a comunicação bloqueante entre componentes remotos. Um dos motivos que nos levou a não optar pelos dois modelos (síncrono e assíncrono) está no fato de que o uso de um modelo bloqueante na arquitetura que planejamos, pode facilmente ocasionar *deadlocks*. A possibilidade de *deadlocks* é decorrente não só da arquitetura do JIC, como também do problema de não componentização de *locks*.

Entretanto, em uma aplicação, existem componentes periféricos, que são aqueles que interagem diretamente com os nós folhas da rede, em geral com a interface gráfica da aplicação. Estes componentes geralmente precisam de comunicação síncrona para interagir com estes nós folhas. Esta necessidade foi identificada dentro do próprio projeto OurGrid, ao realizar a migração para a versão 4.0. Acreditamos que seja possível inserir no JIC um modelo de comunicação síncrono que possibilite a comunicação na periferia da rede de maneira segura. Consideramos esta pesquisa importante, uma vez que é decorrente da necessidade de usuários do JIC.

Bibliografia

[And01] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[And02] Andrew Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 2002.

[Bar05] Bart-Floris Visscher. *Exploring Complexity in Software Systems*. PhD thesis, University of Portsmouth, 2005.

[BGT+01] Francisco Brasileiro, Fabiola Greve, Frederic Tronel, Michel Hurfin, and Jean-Pierre Le Narzul. Eva: An Event-Based Framework for Developing Specialized Communication Protocols. *nca*, 00:0108, 2001.

[BMS02] Marin Bertier, Olivier Marin, and Pierre Sens. Implementation and Performance Evaluation of an Adaptable Failure Detector. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 354–363. IEEE Computer Society, 2002.

[BPS00] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.*, 30(7):775–802, 2000.

[CBA+06] Walfredo Cirne, Francisco Brasileiro, Nazareno Andrade, Lauro Costa, Alisson Andrade, Reynaldo Novaes, and Miranda Mowbray. Labs of the World, Unite!!! *Journal of Grid Computing*, 4(3):225–246, September 2006.

[CBO05] Bruno Catão, Francisco Brasileiro, and Ana Cristina Oliveira. Engineering a Failure Detection Service for Widely Distributed Systems. In *VI Workshop de*

Testes e Tolerância a Falhas, em conjunto com o Simpósio Brasileiro de Redes de Computadores, May 2005., 2005.

[CD05] George Coulouris and Jean Dollimore. Distributed systems: concepts and design. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[CT96] Tushar Chandra and Sam Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225–267, 1996.

[CTA02] Wei Chen, Sam Toueg, and Marcos Kawazoe Aguilera. On the Quality of Service of Failure Detectors. *IEEE Trans. Comput.*, 51(5):561–580, 2002.

[Cur97] David Curtis. Java, RMI and CORBA. Disponível em `http://www.omg.org/library/wpjava.html`, 1997.

[Dav72] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[DCS06] Ayla Dantas, Walfredo Cirne, and Katia Saikoski. Using AOP to Bring a Project Back in Shape: The OurGrid Case. *Journal of the Brazilian Computer Society*, 11(3):21–35, April 2006.

[DDHM02] Xianghua Deng, Matthew B. Dwyer, John Hatcliff, and Masaaki Mizuno. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *ICSE'02: Proceedings of the 24th International Conference on Software Engineering*, pages 442–452. ACM Press, 2002.

[DJMT96] Scott Dawson, Farnam Jahanian, Todd Mitton, and Teck-Lee Tung. Testing of fault-tolerant and real-time distributed systems via protocol fault injection. *ftcs*, 00:404, 1996.

[ECCH00] Dawson R. Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *Proc. 4th Symp. OS Design and Int'l (OSDI 2000)*, pages 1–16. ACM, 2000.

[Ecl06] Eclipse Foundation. Eclipse ide. Disponível em `http://www.eclipse.org`, Acessado em Agosto 2006.

[Eds76] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[Edw06] Edward A. Lee. The Problem with Threads. *Computer*, 39(5):33–42, 2006.

[EGHT94] David Evans, John Guttag, James Horning, and Yang Meng Tan. LCLint: a tool for using specifications to check code. In *SIGSOFT '94: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, pages 87–96, New York, NY, USA, 1994. ACM Press.

[FCO99] Katrina E. Kerry Falkner, Paul D. Coddington, and Michael J. Oudshoorn. Implementing Asynchronous Remote Method Invocation in Java. Technical Report DHPC-072, 1999.

[FDGO99] Pascal Felber, Xavier Defago, Rachid Guerraoui, and Philipp Oser. Failure Detectors as First Class Objects. In *DOA '99: Proceedings of the International Symposium on Distributed Objects and Applications*, page 132. IEEE Computer Society, 1999.

[Fet06] Abe Fettig. *Twisted Network Programming Essentials*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2006.

[For94] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[For05] Bryan Ford. Peer-to-Peer Communication Across Network Address Translators. In *USENIX Annual Technical Conference*, Anaheim, CA, April 2005.

[Fou06] Python Software Foundation. The Python Programming Language. Disponível em `http://www.python.org/`, Acessado em Agosto 2006.

[Fra06] Frank Sauer. Eclipse Metrics Plugin 1.3.6. Disponível em `http://sourceforge.net/projects/metrics`, Acessado em Agosto 2006.

[Gen00] Genesan Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.*, 22(2):416–430, 2000.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Massachusetts, 1994.

[Gro02] William Grosso. *Java RMI*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[HDYK04] Naohiro Hayashibara, Xavier Defago, Rami Yared, and Takuya Katayama. The phi Accrual Failure Detector. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS'04)*, pages 66–78. IEEE Computer Society, 2004.

[Her05] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[HL95] Walter Hursh and Christina Lopes. *Separation of Concerns*. Technical Report NU-CCS-95-03, College of Computer Science, Northeaster University, 1995.

[HP04] David Hovemeyer and William Pugh. Finding concurrency bugs in Java. In *Proceedings of PODC Workshop on Concurrency and Synchronization in Java Programs*, St. John's, Newfoundland, Canada, July 2004.

[IEE06] IEEE. *Distributed Systems Online web site. Message Oriented Middleware section.* Disponível em `http://dsonline.computer.org/middleware/intro_MOM.html`, Acessado em Maio 2006.

[IET06a] IETF. Internet Engineering Task Force web site. Disponível em `http://www.ietf.org`, Acessado em Agosto 2006.

[IET06b] IETF. Requirements for Internet Hosts - Communication Layers. Disponível em `http://www.ietf.org/rfc/rfc1122.txt`, Acessado em Agosto 2006.

[IET06c] IETF. Simple authentication and security layer specification (rfc 4422). Disponível em `http://www.ietf.org/rfc/rfc4422.txt`, Acessado em Agosto 2006.

[IET06d] IETF. Transmission Control Protocol. Disponível em `http://www.ietf.org/rfc/rfc0793.txt`, Acessado em Agosto 2006.

[IET06e] IETF. The transport layer security (tls) protocol version 1.1 (rfc 4346). Disponível em `http://www.ietf.org/rfc/rfc4346.txt`, Acessado em Agosto 2006.

[Jac95] Van Jacobson. Congestion Avoidance and Control. *SIGCOMM Comput. Commun. Rev.*, 25(1):157–187, 1995.

[Jiv06] JiveSoftware. Smack API web site. Disponível em `http://www.jivesoftware.org/smack/`, Acessado em Agosto 2006.

[JSF06] JSF. Jabber software foundation web site. Disponível em `http://www.jabber.org/`, Acessado em Junho 2006.

[JXT06] JXTA. *JXTA Project web site*. Disponível em `http://jxta.org`, Acessado em Agosto 2006.

[Krz02] Krzysztof Palacz and Jan Vitek and Grzegorz Czajkowski and Laurent Daynas. Incommunicado: efficient communication for isolates. In *OOPSLA'02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 262–274, New York, NY, USA, 2002. ACM Press.

[LCBF06] Aliandro Lima, Walfredo Cirne, Francisco Brasileiro, and Daniel Fireman. A Case for Event-Driven Distributed Objects. In *DOA'06: Proceedings of the 8th International Symposium on Distributed Objects and Applications, at OTM Federated Conferences*, October 2006.

[LS88] Barbara Liskov and Liuba Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 260–267, New York, NY, USA, 1988. ACM Press.

[LS05] Steve Loughran and Edmund Smith. *Rethinking the Java SOAP Stack*. Technical Report HPL-2005-83, Hewlett-Packard Bristol Laboratories, May 2005.

[Mah02] Qusay H. Mahmood. Distributed Java Programming with RMI and CORBA. Disponível em `http://developer.java.sun.com/developer/technicalArticles/RMI/rmi_corba/`, January 2002.

[MHC00] Richard Monson-Haefel and David Chappell. *Java Message Service*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.

[Mic06] Sun Microsystems. The java tutorials - Trail: The Reflection API. Disponível em `http://java.sun.com/docs/books/tutorial/reflect/index.html`, Acessado em Agosto 2006.

[OMG98] Object-Management-Group. Corba messaging specification, omg document orbos/98- 05-05 ed., May 1998.

[OMG04] OMG. CORBA 3.0.3, Common Object Request Broker Architecture (Core Specification), 2004-03-01, march 2004.

[OMG06a] OMG. Object Management Group web site. Disponível em `http://www.omg.org/`, Acessado em Agosto 2006.

[OMG06b] OMG. Revised IDL/Java language mapping chapter. Disponível em `http://www.omg.org/cgi-bin/apps/doc?ptc/00-01-08.pdf`, Acessado em Agosto 2006.

[Pie02] Peter R. Pietzuch. Event-Based Middleware: A New Paradigm for Wide-Area Distributed Systems? In *6th CaberNet Radicals Workshop*, February 2002.

[Ram01] Paul Francis Ramakrishna. IPNL: A NAT-extended Internet Architecture. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 69–80, New York, NY, USA, 2001. ACM Press.

[Ram04] Ramesh Chandra and Ryan Lefever and Kaustubh Joshi and Michel Cukier and William Sanders. A global-state-triggered fault injector for distributed system evaluation. *IEEE Trans. Parallel Distrib. Syst.*, 15(7):593–605, 2004.

[RHH85] Jr. Robert H. Halstead. MULTILISP: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, New York, NY, USA, 1995. ACM Press.

[RMI06] RMI. *Java Remote Method Invocation web site*. Disponível em `http://java.sun.com/products/jdk/rmi/`, Acessado em Maio 2006.

[RWB97] Rajeev Raje, Joseph William, and Michael Boyles. An Asynchronous Remote Method Invocation (ARMI) Mechanism for Java. In *ACM Workshop on Java for Science and Engineering Computation*. ACM Press, June 1997.

[SCT95] Gradimir Starovic, Vinny Cahill, and Brendan Tangney. An event based object model for distributed programming. Technical Report TCD-DSG#TCD-CS-95-30, University of Bologna, 1995.

[SP81] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In *Steven S. Muchnick and Neil D. Jones, editors, Program Flow Analysis, chapter 7*, pages 189–234. Prentice Hall, 1981.

[SRSS00] Douglas C. Schmidt, Hans Rohnert, Michael Stal, and Dieter Schultz. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, Inc., 2000.

[Sun90] Vaidy S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990.

[SV99] Douglas Schmidt and Steve Vinoski. Object Interconnections: Programming Asynchronous Method Invocations with CORBA Messaging, 1999.

[Tho76] Thomas J. McCabe. A Complexity Measure. *IEEE Trans. Software Eng.*, 2(4):308–320, 1976.

[Tom75] Raymond S. Tomlinson. Selecting sequence numbers. In *Proceedings of the 1975 ACM SIGCOMM/SIGOPS workshop on Interprocess communications*, pages 11–23. ACM Press, 1975.

[TS06] George Tsirtsis and Pyda Srisuresh. Network Address Translation - Protocol Translation (NAT-PT). Disponível em `http://www.ietf.org/rfc/rfc2766.txt`, Acessado em Agosto 2006.

[WCB01] Matt Welsh, David Culler, and Eric Brewer. Seda: An architecture for well-conditioned, scalable internet services. In *Proceedings of the Eighteenth Symposium on Operating Systems Principles (SOSP-18)*, October 2001.

[WRW96] Ann Wollrath, Roger Riggs, and Jim Waldo. A Distributed Object Model for the Java System. In *2nd Conference on Object-Oriented Technologies & Systems (COOTS)*, pages 219–232. USENIX Association, 1996.

[WWWK94] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. *A Note on Distributed Computing*. Technical Report SMLI TR-94-29, Sun Microsystems Labs, November 1994.

[XMP06] XMPP. Extensible messaging and presence protocol specification. Disponível em `http://www.xmpp.org/specs/`, Acessado em Junho 2006.