Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Informática

Dissertação de Mestrado

Provendo segurança e uma semântica de falhas consistente para a comunicação de objetos assíncronos distribuídos

Rodrigo de Almeida Vilar de Miranda

Campina Grande, Paraíba, Brasil
Agosto - 2010

Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Informática

Provendo segurança e uma semântica de falhas consistente para a comunicação de objetos assíncronos distribuídos

Rodrigo de Almeida Vilar de Miranda

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

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Redes e Sistemas Distribuídos

Francisco Vilar Brasileiro
(Orientador)

Campina Grande, Paraíba, Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

M672p Miranda, Rodrigo de Almeida Vilar de.
Provendo segurança e uma semântica de falhas consistente para a comunicação de objetos assíncronos distribuídos /Rodrigo de Almeida Vilar de Miranda. — Campina Grande, 2010.
92 f.: il.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.
Orientador: Profº. Francisco Vilar Brasileiro
Referências.

1. Sistemas de Processamento Distribuído. 2. Tolerância a Falhas. 3. Segurança. I. Título.

CDU 004.75(043)

"PROVENDO SEGURANÇA E UMA SEMÂNTICA DE FALHAS CONSISTENTE PARA A COMUNICAÇÃO DE OBJETOS ASSÍNCRONOS DISTRIBUÍDOS"

RODRIGO DE ALMEIDA VILAR DE MIRANDA

DISSERTAÇÃO APROVADA EM 10.08.2010

FRANCISCO VILAR BRASILEIRO, Ph.D
Orientador(a)

ANTÔNIO TADEU AZEVEDO GOMES, Dr.
Examinador(a)

LÍVIA MARIA RODRIGUES SAMPAIO CAMPOS , D.Sc
Examinador(a)

CAMPINA GRANDE - PB

Resumo

O desenvolvimento de sistemas distribuídos é uma atividade muito complexa, mas que pode ser facilitada com o uso de sistemas de *middleware* adequados. Este trabalho abrange o nicho dos sistemas distribuídos *peer-to-peer* em Java, cujos nós precisam manter diversas conexões abertas simultaneamente, pois não conhecemos sistemas de *middleware* que suportem plenamente as necessidades deste nicho. Com a finalidade de avaliarmos esta lacuna, elicitamos sete requisitos não-funcionais para o nicho estudado: detecção de falhas por parada, detecção de falhas por perda de mensagens, concorrência, facilidade de programação, segurança, suporte a conectividade parcial e integração com a linguagem Java. Dentre cinco sistemas de *middleware* avaliados nenhum atendeu a todos os requisitos não-funcionais, o que nos motivou a desenvolver um novo sistema de *middleware*, chamado Commune, que atendesse a todos os requisitos. Desse modo, o Commune poderia suportar eficazmente o desenvolvimento de sistemas para o nicho estudado neste trabalho. Em relação aos sistemas avaliados, o Commune provê duas contribuições principais. Primeiramente, realizamos uma pesquisa bibliográfica na área de segurança para sistemas distribuídos, a partir da qual implementamos mecanismos de segurança para o Commune, baseados em (i) autenticação com pares de chaves assimétricas e (ii) certificação X.509. Em segundo lugar, definimos os requisitos de uma semântica de detecção de falhas abrangente, capaz de detectar falhas por parada dos nós e falhas por omissão devido a perda de mensagens no canal de comunicação. Modelamos um protocolo de conexão para o Commune, utilizando uma máquina de estados para representar o estados dos nós, e provamos que o modelo proposto atende aos requisitos da semântica de detecção de falhas abrangente. Foi necessário o uso de técnicas de verificação formal para provar que o protocolo não possui *deadlocks*. Além disso, apresentamos a arquitetura e os detalhes de implementação do Commune. Por fim, avaliamos o desempenho do Commune, que se mostrou viável na comparação com Java RMI, e executamos testes do Commune em ambiente de produção, que ocorreram com sucesso.

Abstract

The development of distributed systems is a very complex activity, but it can be aided by appropriate middleware platforms. This work encompasses the niche of peer-to-peer distributed systems developed in Java, whose nodes need to maintain several connections opened simultaneously, because, as far as we concern, there is no middleware which supports all the demands of this niche. In order to evaluate that gap, we have elicited seven non-functional requirements for systems in the work niche: crash failure detection, message loss failure detection, concurrence, programming easiness, security, partial connectivity support and integration with the Java programming language. None of the five middleware that we evaluated met all the non-functional requirements, so we were motivated to develop a new middleware, called Commune, which would meet all these requirements. As a result, the Commune middleware could aid the development of systems in the niche of this work. In comparison with the evaluated middleware, Commune has two main contributions. Firstly, we carried out a bibliographic research in the extent of security for distributed systems, in order to implement security for Commune. We have created two mechanisms based on asymmetric key pair authentication and X.509 certification. In second place, we have defined the requirements for a comprehensive failure detection semantic, able to detect node crash failures and omission failures due to message lost in the communication channel. We have modelled a connection protocol for Commune, using a state machine to illustrate the state of Commune nodes, and we have proven that the proposed model meet all the requirements of the comprehensive failure detection semantic. It was necessary to use formal verification techniques, in order to prove that the connection protocol does not contain deadlocks. Moreover, we have shown the Commune architecture and implementation details. At last, we have evaluated the Commune performance, which is feasible when it is compared to Java RMI, and we have executed test in a production environment, with successful results.

Agradecimentos

Em primeiro lugar, gostaria de agradecer ao professor Francisco Brasileiro não só pela orientação neste mestrado, mas por todo o investimento feito ao me trazer para a equipe de desenvolvimento do OurGrid. Nestes três anos, foram muitos puxões de orelha e incentivos que me fizeram crescer como cientista e como pessoa.

À minha amada esposa, Mariana, o reconhecimento pelo incentivo e pelas renúncias em prol deste nosso objetivo comum.

Aos meus pais, por fomentarem sempre o sonho de ser um mestre.

A todos aqueles que passaram pela equipe de desenvolvimento do OurGrid e me ajudaram direta ou indiretamente neste trabalho. Principalmente, Abmar e Adabriand, meus braços direito e esquerdo, sem os quais não teria conseguido os resultados obtidos nos testes de produção.

Meu mais intenso agradecimento a Deus, que foi meu maior auxílio dos momentos em que nada parecia dar certo, pois se cumpriram as palavras do provérbio: *reconhece-o em todos os teus caminhos e ele endireitará as tuas veredas*.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

O desenvolvimento de sistemas distribuídos para Internet é uma atividade mais complexa do que a construção de sistemas locais, também chamados de *standalone*. Em um sistema local, os componentes executam em apenas um processo e dentro do mesmo espaço de endereçamento, portanto o acesso entre eles é direto. Neste caso, o programador não precisa se preocupar, por exemplo, com a indisponibilidade de um componente de software local, nem com aspectos de segurança nas invocações de funções locais. Além disso, se não forem utilizados vários fluxos de execução (*Threads*), também não será preciso tratar o acesso concorrente aos recursos do sistema.

Por outro lado, em um ambiente distribuído o programador precisa lidar com características mais complexas, que tornam o desenvolvimento do software mais difícil. O Laboratório de Sistemas Distribuídos (LSD) da UFCG tem trabalhado na pesquisa e desenvolvimento de sistemas distribuídos desde 1995. Após todos esses anos de experiência no LSD, nós podemos elencar as características mais complexas no desenvolvimento desse tipo de sistemas, no nosso ponto de vista: tolerância a falhas, concorrência, segurança e conectividade parcial.

Diversas plataformas de comunicação, denominadas sistemas de *middleware*, foram criadas com a finalidade de facilitar o desenvolvimento de sistemas distribuídos. O sistema de *middleware* é um software responsável por prover os serviços básicos de comunicação, que implementam alguns requisitos não-funcionais (qualidade) dos sistemas distribuídos. Dessa forma, o programador pode concentrar seus esforços na implementação dos requisitos funcionais (lógica de negócio). Existem diversos tipos de sistemas distribuídos e cada um tem seus requisitos não-funcionais específicos. Portanto, um sistema de *middleware* que se

adapta bem a um determinado tipo de sistema distribuído pode ser totalmente inapropriado para outro tipo.

Em 2002, iniciou-s no LSD o projeto OurGrid, uma grade computacional *peer-to-peer* de livre entrada [7]. As primeiras versões do OurGrid foram lançadas a partir de 2004 utilizando como sistema de *middleware* a tecnologia Java RMI [37]. No entanto, houve uma dificuldade considerável na programação do sistema, à medida que a quantidade de fluxos de execução concorrentes crescia. Isto nos levou a questionar se Java RMI seria o sistema de *middleware* ideal para o OurGrid.

Em uma pesquisa mais genérica, identificamos que não apenas o OurGrid, mas todo o nicho dos sistemas distribuídos *peer-to-peer - P2P*, cujos nós[1] precisam manter muitas conexões abertas simultaneamente, não possui um sistema de *middleware* para plataforma Java bem adaptado aos seus requisitos não-funcionais. O que nos motivou a investigar mais profundamente esse nicho de sistemas.

Na próxima seção, explicaremos cada uma das características que dificultam o desenvolvimento de sistemas distribuídos. Posteriormente, discorreremos sobre a evolução dos sistemas distribuídos e delimitaremos o escopo deste trabalho. No final do Capítulo, apresentaremos a estrutura da dissertação.

1.1 Características dos sistemas distribuídos

Nesta seção, discorreremos sobre algumas características dos sistemas distribuídos que tornam o seu desenvolvimento mais complexo que o desenvolvimento de sistemas *standalone*.

Sistemas distribuídos podem sofrer **falhas parciais**, quando um subconjunto dos seus nós se torna indisponível, por isto esses sistemas devem ser capazes de tratar as falhas, reagindo de alguma forma consistente. Como exemplos de reações às falhas, podemos citar (i) a substituição dos nós indisponíveis, (ii) a postergação da computação ou (iii) a exibição do erro para o usuário. Um outro tipo de falha, que pode ocorrer no meio de comunicação, é a **perda de mensagens** entre dois nós. Neste caso, a comunicação entre os nós pode se tornar inconsistente. Portanto o sistema deve detectar perdas de mensagens, sinalizando aos nós para se sincronizarem novamente.

[1]Neste trabalho utilizaremos o termo nó para nos referirmos de forma genérica aos componentes de um sistema distribuído. Esses podem ser processadores, processos, objetos, etc.

Os processos de um sistema distribuído, localizados na mesma máquina ou em máquinas distintas, podem concorrer pelo acesso a recursos compartilhados. Durante a construção do sistema, o programador precisa estar ciente dessa **concorrência**. Ele deve proteger as regiões críticas do seu código e evitar algoritmos que possam entrar em impasse (*deadlock*) ou inanição (*livelock*). Um sistema distribuído entra em impasse quando todos os seus processos estão bloqueados, esperando por alguma informação de outro processo e não conseguem avançar para um próximo estado [20]. No estado de inanição, o sistema como um todo não consegue evoluir, embora os processos individualmente não estejam bloqueados [40].

Se as mensagens do sistema distribuído trafegam pela Internet, o arquiteto de software precisa prever o uso de mecanismos de **segurança**, a fim de proteger as informações sensíveis e preservar o bom funcionamento do sistema. Se o sistema não for seguro, os usuários maliciosos da rede poderão invadir os bancos de dados, visualizar dados sigilosos, alterar informações valiosas e forjar identidades falsas. Além disto, o sistema pode ser alvo de ataques de negação de serviço, tornando-o indisponível.

A implantação de alguns tipos de sistemas distribuídos pode ser prejudicada quando o meio de comunicação apresenta **conectividade parcial**, como ocorre nas redes privadas e nas redes protegidas por *firewalls* [9]. De forma simples, a conectividade parcial pode ser descrita como o cenário onde um nó A pode abrir conexão com um nó B, mas o nó B não pode iniciar conexão com o nó A. Nos sistemas distribuídos com arquitetura cliente–servidor, essa limitação não impede o funcionamento do software, pois apenas o cliente precisa abrir conexão com o servidor. Por outro lado, os sistemas *peer-to-peer*, onde um mesmo nó pode assumir o papel de cliente e servidor simultaneamente, precisam utilizar alguma solução alternativa para ultrapassar as barreiras de conectividade parcial.

1.2 Evolução dos sistemas distribuídos

Esta seção mostra resumidamente a evolução na arquitetura dos sistemas distribuídos até o advento dos sistemas *peer-to-peer*.

A introdução das redes de computadores, nos anos 70, viabilizou o desenvolvimento dos primeiros sistemas distribuídos [2]. Inicialmente, esses sistemas eram compostos por terminais que se conectavam a um computador central, apenas para entrada e saída de dados.

As conexões entre os nós desse tipo de sistema criam uma topologia no formato estrela, na qual o servidor representa o centro e os terminais são as pontas da estrela.

Posteriormente, com o surgimento dos computadores pessoais, foram desenvolvidos sistemas distribuídos com a arquitetura cliente-servidor [32]. A topologia continuava no formato estrela, mas as estações clientes possuíam a lógica de apresentação e de negócio. O servidor era responsável pelo acesso e persistência centralizados dos dados. A escalabilidade da arquitetura cliente-servidor era limitada à quantidade de conexões que o servidor poderia manter simultaneamente. Portanto, quando esses sistemas precisavam atender centenas de usuários, sua desempenho era bastante degradada.

Alguns anos depois, surgiram os sistemas multi-camadas [32], nos quais o servidor central era implementado por diversas instâncias de servidores de dados (acesso e persistência dos dados) e servidores de aplicação (lógica de negócio). Os clientes utilizavam navegadores web para acessar os servidores de aplicação. Embora a principal motivação para o surgimento desses sistemas tenha sido facilitar a atualização de software nos clientes, essa arquitetura também melhorou a escalabilidade dos sistemas, permitindo atender milhares de usuários simultaneamente, pois os bancos de dados precisam manter conexões apenas com os servidores de aplicação. Estes, por sua vez, mantém um *pool* de processos ou *Threads* que permitia o atendimento de centenas de requisições *HTTP* dos usuários em cada servidor.

Na década de 90, surgiram os primeiros sistemas *peer-to-peer* para Internet [29], que possibilitavam a troca direta de mensagens entre os usuários, sem precisar trafegá-las por um servidor central. Esse modelo representa uma evolução considerável na topologia dos sistemas distribuídos, pois cada nó poderia assumir o papel de cliente e servidor simultaneamente, se conectando com qualquer outro nó. Neste caso, as conexões do sistema não são representadas como uma estrela. Em vez disto, a topologia poderia assumir a forma de qualquer grafo. Como consequência desse modelo, a programação concorrente, utilizada para manter diversas conexões simultaneamente, não está mais isolada no servidor central. Todos os nós do sistema podem se conectar a vários outros nós. Alguns exemplos de sistemas distribuídos P2P incluem BitTorrent[2], Gnutella[3] e OurGrid[4].

Neste mesmo período, a OMG lançou a especificação CORBA [28] com o objetivo de

[2]http://www.bittorrent.com
[3]http://www.gnutella.com
[4]http://www.ourgrid.org

promover a interoperabilidade de software implantado sobre plataformas de hardware ou sistemas operacionais diferentes. Além disto, componentes escritos em linguagens diferentes também poderiam se comunicar. Dessa forma, CORBA prometia ser a tecnologia de última geração para o comércio eletrônico, o que não aconteceu devido a grandes mudanças que aconteceram no fim dos anos 90, dentre as quais destaca-se o advento da Web e da linguagem Java [16; 17]. Enquanto a implementação das aplicações CORBA era dificultada pela complexidade e inconsistência da API[5], o modelo de componentes Java, *Enterprise JavaBeans - EJB*, era mais simples e facilitava a programação dos componentes distribuídos. Além disto, havia limitações na cooperação de CORBA com a Web, portanto as empresas começaram a desenvolver suas plataformas de comércio eletrônico baseadas em servidores Web, HTTP, Java e EJB. Assim sendo, Java é atualmente uma das tecnologias mais utilizadas para o desenvolvimento de sistemas distribuídos implantados na Internet.

1.3 Sistemas P2P com muitas conexões simultâneas

Cada sistema de *middleware* possui características que o tornam mais adequado para um determinado tipo de sistema distribuído. Portanto, a escolha do sistema de *middleware* que deve ser utilizado como plataforma de desenvolvimento para um determinado sistema distribuído deve levar em consideração (i) os requisitos não-funcionais do sistema distribuído a ser desenvolvido e (ii) até que ponto o sistema de *middleware* atende esses requisitos. Por exemplo, se um sistema distribuído demanda fortes requisitos de segurança, e um sistema de *middleware* não possui integração com nenhum mecanismo de segurança, então provavelmente esse sistema *middleware* não será uma boa escolha para o sistema distribuído em questão.

O escopo desta dissertação se restringe aos sistemas distribuídos P2P implementados em Java, cujos nós precisam manter muitas conexões abertas simultaneamente. Um exemplo desse tipo de sistemas é o OurGrid, uma plataforma para computação em grade *peer-to-peer* de livre entrada. O OurGrid é composto por três tipos principais de componentes: o *Broker* é a interface do usuário por onde ele submete tarefas para a grade; o *Worker* é o agente instalado em cada máquina da grade que recebe as tarefas do usuário e as executa

[5] API - do inglês, Application Programming Interface

remotamente; e o *Peer* é o gerente de cada site OurgGrid, que controla os *Brokers* e *Workers* do site local e se conecta aos *Peers* dos sites remotos para requisitar mais *Workers*.

Em alguns cenários do OurGrid, os *Brokers* podem manter centenas de conexões abertas simultaneamente com vários *Workers* da grade, tanto locais como remotos. Não é prioritário que a comunicação *Broker - Worker* possua alta vazão de dados, pois geralmente o tempo de execução das tarefas é várias ordens de grandeza maior do que o tempo de transferência de arquivos. Todavia, o *Broker* precisa monitorar constantemente cada *Worker*, pois se este falhar, a tarefa deverá ser submetida para outro computador. A Figura 1.1 mostra um exemplo do cenário onde existem muitas conexões *Broker - Worker* simultâneas.

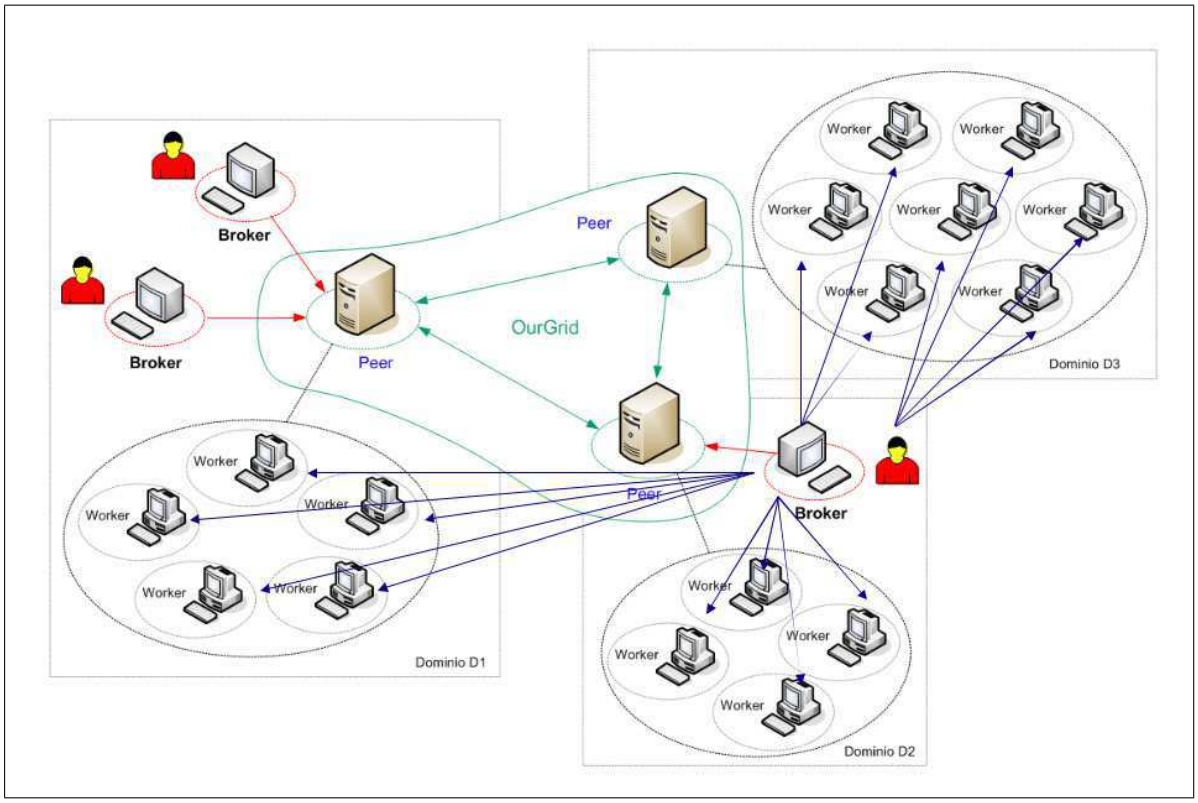

Figura 1.1: Exemplo de muitas conexões simultâneas no OurGrid

Como pode ser visto no Capítulo 2, os sistemas distribuídos com poucas conexões possuem várias opções de sistemas de *middleware* para o seu desenvolvimento. No entanto, quando os nós dos sistemas distribuídos precisam manter muitas conexões simultâneas, surgem novos requisitos não-funcionais que não são completamente suportados pelos sistemas de *middleware* conhecidos para a plataforma Java. No caso do OurGrid, versão 3.0, que utilizava Java RMI [37] como plataforma de comunicação, foram encontradas dificuldades para

programar e testar o código com múltiplos fluxos de execução concorrentes, principalmente na comunicação *Broker - Worker*. Por essa razão, o OurGrid enfrentou problemas de escala que limitavam o tamanho da grade computacional a poucas centenas de nós.

Diante do exposto, elicitamos os requisitos não-funcionais de um sistema distribuído P2P, cujos nós precisam manter diversas conexões abertas simultaneamente. Esses requisitos serviram, inicialmente, para avaliar os sistemas de *middleware* conhecidos, mostrando as suas lacunas. Posteriormente, os requisitos não-funcionais foram utilizados para nortear a construção de um novo sistema de *middleware* que os atendesse plenamente.

1.3.1 Requisitos não-funcionais

A seção 1.1 descreve algumas características que, segundo a nossa experiência, tornam o desenvolvimento de sistemas distribuídos mais complexo que o de sistemas *standalone*. Portanto, iniciamos listando os requisitos não-funcionais[6] que o sistema de *middleware* deve atender, referenciando os mecanismos que facilitem a implementação dessas características.

Requisito 1 - Detecção de falha dos nós: O sistema de *middleware* será capaz de detectar a falha de um nó ou de um conjunto de nós do sistema distribuído, notificando a falha aos nós interessados, para que estes possam tratá-la conforme a lógica da aplicação definir.

Requisito 2 - Detecção de perda de mensagens: O sistema de *middleware* será capaz de detectar a perda de mensagens no canal de comunicação, notificando a falha aos nós interessados, para que estes possam tratá-la conforme a lógica da aplicação definir.

Requisito 3 - Concorrência no middleware: O sistema de *middleware* deve estabelecer um protocolo de comunicação robusto que não possua *deadlocks* nem *livelocks*, exceto nos casos onde há erro na aplicação desenvolvida sobre o sistema de *middleware*.

Requisito 4 - Facilidade de programação: O sistema de *middleware* proverá uma API para comunicação remota concorrente, que facilite a programação e os testes de componentes capazes de se comunicar com um grande número de nós remotos simultaneamente.

[6]Cada requisito elicitado nesta subseção possui, além da sua descrição, um título curto pelo qual será referenciado no restante desta dissertação

Requisito 5 - Segurança: O sistema de *middleware* deve prover mecanismos de segurança, a fim de proteger o sistema distribuído dos ataques que podem ser efetuados através da Internet.

Requisito 6 - Conectividade parcial: A arquitetura do sistema de *middleware* possibilitará a inclusão de nós com conectividade parcial ao sistema distribuído, mesmo que estejam protegidos por *firewalls* ou pertençam a redes privadas.

O último requisito a ser elicitado tem a ver com a necessidade do sistema de *middleware* expor uma API para o programador do sistema distribuído. De fato, a responsabilidade do sistema de *middleware* é a provisão de alguns requisitos não-funcionais do sistema. O restante dos componentes deve ser implementado pelo programador, utilizando uma linguagem de programação que esteja bem integrada ao sistema de *middleware*. Nesse sentido, especificamos que o sistema de *middleware* deve prover uma boa integração com a linguagem de programação orientada a objetos Java, a fim de permitir a modularização dos sistemas em componentes de software independentes. Dessa forma, se um sistema distribuído for grande e demandar uma equipe de programadores para ser construído, cada programador poderá focar no desenvolvimento de um componente e não precisará estar ciente dos detalhes internos dos componentes adjacentes.

Requisito 7 - Orientação a objetos: O sistema de *middleware* proverá uma API bem integrada com a linguagem orientada a objetos Java.

1.3.2 Commune

Os sete requisitos não-funcionais, elicitados na subseção anterior, foram utilizados para verificar se os sistemas de *middleware* existentes atendem às necessidades dos sistemas distribuídos P2P implementados em Java, cujos nós precisam manter muitas conexões abertas simultaneamente. Como nenhum dos sistemas de *middleware* conhecidos atendeu a todos os requisitos, decidimos implementar um novo sistema de *middleware*, chamado **Commune**, para dar suporte ao desenvolvimento desse tipo de sistemas distribuídos.

O **Commune** se baseou no *JIC - Java Internet Communication* [24], contribuindo com a adição de dois novos componentes, a fim de atender a todos os sete requisitos elicitados

nesta seção: (i) um portifólio de mecanismos de segurança e (ii) uma semântica de detecção de falhas consistente. O **Commune** foi testado em produção na versão 4 do OurGrid e se mostrou robusto e escalável.

1.4 Estrutura da dissertação

O Capítulo corrente introduziu o contexto do trabalho e o problema chave desta dissertação. O restante da dissertação está organizado da seguinte forma:

Os trabalhos relacionados são descritos no Capítulo 2, onde os principais sistemas de *middleware* conhecidos serão avaliados em relação aos sete requisitos não-funcionais elicitados na introdução. Nesse Capítulo, identificamos que nenhuma das tecnologias avaliadas atendeu a todos os requisitos, o que motiva o desenvolvimento de um novo sistema de *middleware* para esse nicho.

Nesse contexto, o Capítulo 3 apresenta o Commune, um sistema de *middleware* para comunicação assíncrona de objetos distribuídos, que atende a todos os requisitos não-funcionais elicitados na introdução. A criação de um portifólio de mecanismos de segurança flexível é a primeira contribuição do Commune. Além disto, este Capítulo mostra a modelagem de um protocolo de conexão que torna o Commune capaz de detectar e tratar a perda de mensagens no canal de comunicação.

Os detalhes de implementação do Commune são descritos no Capítulo 4, que também demonstra alguns exemplos de uso do sistema de *middleware*.

O Capítulo 5 retrata os procedimentos utilizados para validar e testar o sistema de *middleware* Commune. O modelo do protocolo de conexão do Commune foi validado em relação ao atendimento dos predicados de uma semântica de detecção de falhas abrangente. Em um desses predicados foi necessário o uso de ferramentas formais. Além disto, o Commune passou por uma avaliação de desempenho da solução, testes automáticos e um experimento de uso em ambiente de produção.

A conclusão do trabalho se encontra no Capítulo 6. Por fim, o Apêndice A contém o código fonte do modelo que foi utilizado para verificar formalmente o protocolo de conexão do Commune.

Capítulo 2

Trabalhos relacionados

Neste Capítulo, avaliamos alguns dos sistemas de *middleware* conhecidos, verificando se eles atendem aos requisitos não-funcionais dos sistemas P2P em Java, cujos nós precisam manter muitas conexões simultaneamente. Esses requisitos foram elicitados na seção 1.3.1.

Dividimos os sistemas de *middleware* conhecidos em três grupos: (i) os que se baseiam no modelo de objetos distribuídos, (ii) sistemas de *middleware* orientados a mensagens e (iii) sistemas de *middleware* híbridos, que também podem ser chamados de objetos assíncronos. Além disso, fizemos uma pesquisa por soluções disponíveis em linguagens de programação que não fossem Java, somente para referenciá-las neste trabalho. Posteriormente, sintetizamos a avaliação dos sistemas de *middleware* para Java em um quadro comparativo e identificamos que nenhum deles atende a todos os requisitos não-funcionais. Essa lacuna nos motivou para a criação de um novo sistema de *middleware*, que será apresentado no Capítulo 3, capaz de atender a todos esses requisitos.

2.1 Modelo de objetos distribuídos

O modelo de objetos distribuídos é uma adaptação do paradigma de programação orientado a objetos para sistemas distribuídos. A troca de mensagens entre os nós é encapsulada pelo sistema de *middleware*, de forma que os objetos remotos, localizados em diferentes processos, podem se comunicar com invocações de métodos, tendo a impressão que estão no mesmo espaço de endereçamento.

Esse modelo representa um grande progresso para o desenvolvimento de sistemas distri-

buídos [9], pois as linguagens orientadas a objeto permitem a modularização dos sistemas, organizando seu código em componentes independentes e reduzindo a sua complexidade. Desse modo, o programador pode desenvolver um componente sem se preocupar com os detalhes internos dos componentes adjacentes.

A fim de emular o comportamento de objetos locais, a maior parte das implementações do modelo de objetos distribuídos utiliza uma abordagem síncrona. O fluxo de execução (*Thread*)[1], no processo do cliente, envia uma mensagem para o fluxo do servidor remoto e se bloqueia até que a mensagem de resposta retorne, com um resultado ou uma exceção.

A Figura 2.1 mostra um diagrama de sequência para ilustrar a interação entre dois nós, no contexto de uma chamada remota síncrona. Nesse diagrama, a chamada remota é representada pela mensagem $ping()$ e a linha tracejada representa o retorno da chamada. Essa abordagem é uma boa adaptação entre a programação remota e a programação local, pois a chamada remota aparenta ser apenas mais um passo na execução sequencial do cliente. A desvantagem é que os recursos alocados para o fluxo de execução do cliente ficam ociosos durante a invocação remota.

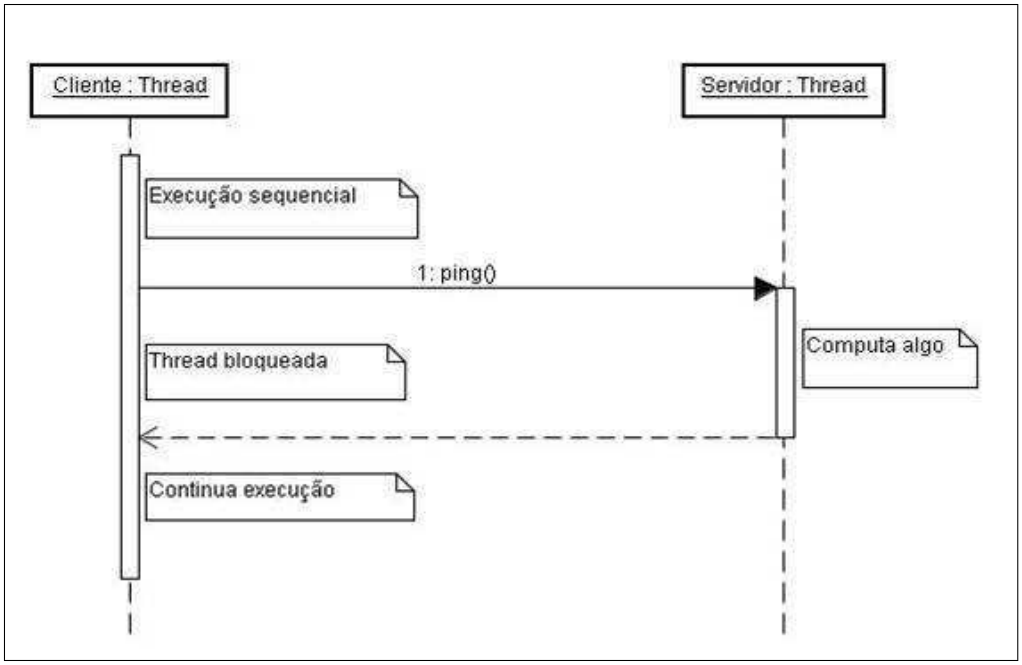

Figura 2.1: Abordagem síncrona

[1]Deste ponto em diante, utilizaremos apenas os termos fluxo ou fluxo de execução, para nos referirmos a um fluxo de execução em um processo.

Em relação à detecção de falhas, Chandra e Toueg [6] dizem que em sistemas síncronos é possível implementar um detector de falhas perfeito, no qual os nós não são suspeitos erroneamente e os processos falhos são suspeitos.

Todavia, a abordagem síncrona pode não ser ideal para os sistemas distribuídos P2P que mantêm muitas conexões abertas [24], pois será necessário utilizar *locks* em muitos trechos de código, a fim de sincronizar corretamente o estado em memória para diversos fluxos ativos. Desse modo, o código fonte se torna mais complexo e difícil de testar, indo de encontro ao Requisito 4 - Facilidade de programação. Além disto, a programação com múltiplos fluxos reduz o encapsulamento dos objetos, porque os objetos precisarão conhecer os detalhes internos (*locks*) uns dos outros.

Dentre as tecnologias que utilizam a abordagem síncrona, decidimos avaliar duas delas em relação ao atendimento dos requisitos não-funcionais definidos na seção 1.3.1. Primeiramente, estudamos **CORBA** [28], por se tratar de uma especificação padrão para o modelo de objetos distribuídos. Posteriormente, avaliamos **Java RMI** [37], um dos sistemas de *middleware* baseados no modelo de objetos distribuídos mais utilizados na atualidade. Além dessas plataformas, poderíamos citar .Net Remoting [25] e Web Services [36]. .Net Remoting que é utilizado para implementação de sistemas distribuídos na plataforma .Net. Todavia, não avaliaremos esse sistema de *middleware* pois o mesmo possui características bem semelhantes ao Java RMI. Os Web Services possuem características parecidas com as de Java RMI, devido ao fato de utilizarem o protocolo HTTP para comunicação remota e, portanto, precisarem de acesso direto entre os nós.

2.1.1 CORBA

Definido pela *Object Management Group (OMG)*, o padrão **CORBA** – *Commom Object Request Broker Architecture* [28] – é uma especificação para o modelo de objetos distribuídos. O objetivo de CORBA é permitir a construção de aplicações completamente distribuídas, com partes de software interoperando com independência de plataforma, sistema operacional, linguagem de programação e protocolo. Por essa razão, em vez de prover componentes de software concretos, CORBA define uma especificação genérica, chamada *Interface Definition Language (IDL)*, que pode ser mapeada em linguagens reais. Além do modelo de objetos distribuídos, CORBA também provê um serviço de troca de mensagens para comu-

nicação assíncrona.

Segundo Henning [17], CORBA chegou a ser um sistema de *middleware* bem popular, mas não conseguiu atingir os objetivos de interoperabilidade, pois as implementações de CORBA são incompatíveis entre si. Posteriormente, CORBA se tornou uma tecnologia de nicho e seu uso está em declínio. Henning elenca alguns problemas de CORBA que determinaram o seu fracasso: API complexa; debilidades no projeto de interfaces e no mapeamento para linguagens; lacunas de segurança que permitem ataques do tipo *man-in-the-middle*; a necessidade de abrir uma porta no *firewall* para cada serviço CORBA, o que pode ir de encontro às políticas de segurança das corporações; ausência de um mecanismo de versionamento, obrigando todos os componentes de um sistema a serem atualizados ao mesmo tempo, o que pode ser tecnicamente inviável; entre outras deficiências.

No contexto deste trabalho, as principais deficiências de CORBA, são (i) as lacunas de segurança (Requisito 5) e (ii) a necessidade de abrir portas no *firewall*, dificultando o atendimento ao Requisito 6 - Conectividade parcial. A OMG define uma especificação para tolerância a falhas, chamada FT-CORBA, capaz de detectar de falhas por parada dos nós [4]. Pelo fato das conexões entre os nós serem diretas, o Requisito 2 - Detecção de perda de mensagens - pode ser atendido pelo protocolo sobre o qual os componentes CORBA estão se comunicando, como por exemplo o TCP/IP.

2.1.2 Java RMI

Uma das tecnologias de objetos distribuídos mais difundidas é o Java RMI [37]. Essa tecnologia utiliza a abordagem síncrona (bloqueante) para a comunicação remota entre objetos. Devido à sua API bloqueante para as chamadas remotas, o código fonte com RMI possui boa legibilidade. Além disto, o tratamento de falhas de comunicação é facilitado, pois o programador precisa apenas tratar a exceção *RemoteException*. Neste caso, tanto na falha por parada dos nós quanto na falha de omissão decorrente da perda de mensagens, o sistema de *middleware* tenta reenviar a mensagem por um determinado tempo e, se após esse tempo ele não obtiver sucesso nessa tentativa, a exceção é lançada no cliente. Por outro lado, o uso da abordagem síncrona torna a sincronização dos fluxos mais complexa, pois reduz o encapsulamento dos objetos e dificulta a realização de testes. Consequentemente, a programação concorrente torna-se mais difícil.

Java RMI não é apropriado para sistemas onde cada nó precisa se comunicar com muitos nós simultaneamente, pois é necessário manter um fluxo em memória para cada conexão aberta. Logo, pode haver um estouro de memória quando forem abertas muitas conexões simultâneas. Além disto, JAVA RMI não possui uma abordagem simples para o tratamento de obstáculos na conectividade da Internet (Requisito 6 - Conectividade parcial), pois exige o acesso direto entre os nós em comunicação. Esse acesso direto torna-se possível se houver (i) mudanças na política de administração da rede, como aberturas de portas no *firewall*, ou (ii) se for utilizado tunelamento HTTP para que a conexão trafegue pela porta 80, que geralmente é aberta nos *firewalls*. Porém, esta última solução adiciona carga extra na comunicação, além de impactar as políticas de segurança da corporação. Em relação ao suporte à segurança (Requisito 5), a troca de mensagens RMI pode ser criptografada se o tipo de *socket* criado para a conexão remota utilizar *SSL - Secure Sockets Layer*. Todavia, não encontramos na literatura nenhuma referência para uma integração de JAVA RMI com uma infra-estrutura de certificação, que garanta a identidade dos nós participantes do sistema.

2.2 Sistemas de middleware orientados a mensagens

Os *MOMs* (do inglês *Message-Oriented Middleware*) são plataformas para comunicação em grupo no padrão *publish/subscribe* [34; 5]. Esse tipo de sistema de *middleware* utiliza uma abordagem assíncrona para as invocações remotas. Cada nó do sistema possui uma fila de mensagens, que é povoada quando o nó recebe mensagens oriundas de nós remotos. Posteriormente, um fluxo de execução consome uma mensagem da fila e a converte em execução da lógica da aplicação.

Nessa abordagem, ao invés de bloquear durante uma invocação remota, o cliente envia uma mensagem para o servidor e libera o fluxo de execução para executar a próxima mensagem da fila. Quando o servidor terminar a computação, envia uma mensagem de resposta para o cliente, que é processada assincronamente. Essa interação assíncrona está exemplificada no diagrama de sequência da Figura 2.2.

Se um nó do sistema distribuído possuir apenas um fluxo de execução consumindo as mensagens da fila, então as mensagens podem ser executadas sem concorrência. Desse modo, a implementação da aplicação é mais simples e pode-se preservar o encapsulamento

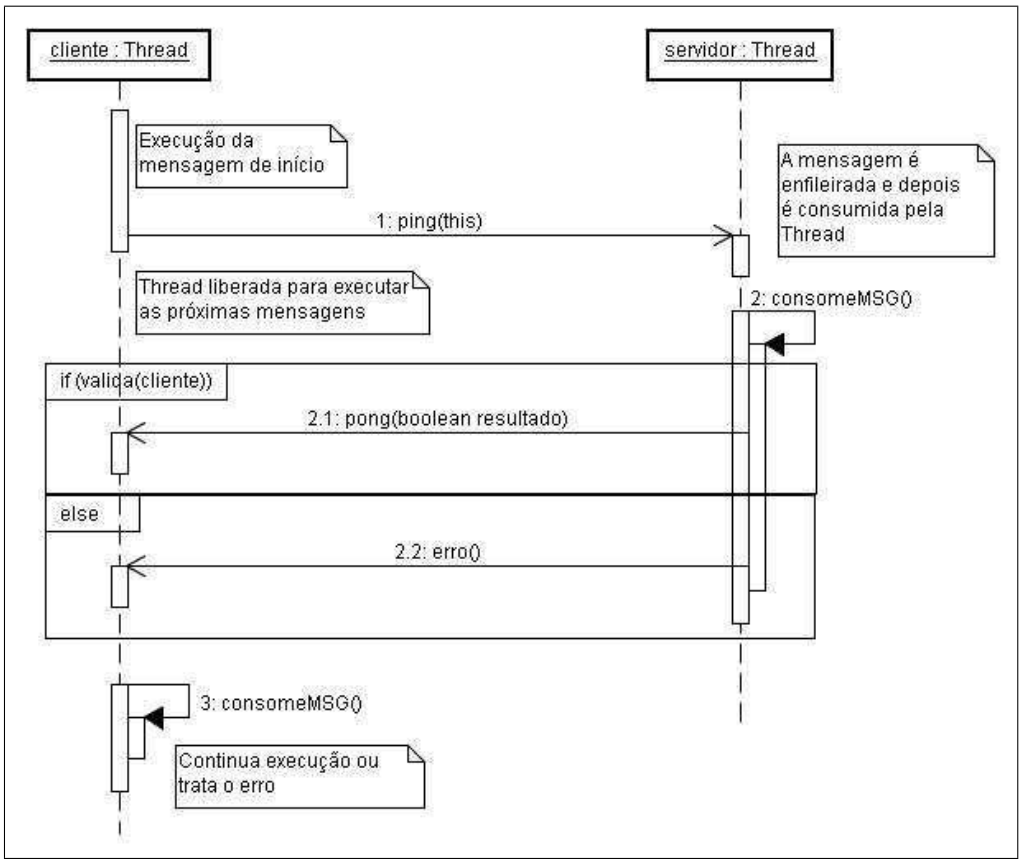

Figura 2.2: Abordagem assíncrona

dos objetos (Requisito 4 - Facilidade de programação).

Nos MOMs, a publicação e o consumo de eventos usam primitivas de baixo nível semelhantes às *send* / *receive* utilizadas para comunicação via troca de mensagens. Portanto, a integração com a linguagem de programação orientada a objetos não é tão forte quanto no modelo de objetos distribuídos (Requisito 7 - Orientação a objetos). Essas primitivas de baixo nível não permitem a verificação de tipo nas iterações remotas. Portanto, muitos erros que poderiam ser detectados em tempo de compilação aparecerão apenas em tempo de execução. Além disto, o código fonte assíncrono é mais complexo do que o código síncrono. Na abordagem síncrona, a programação flui sequencialmente, após a invocação remota. Nos sistemas assíncronos, o código não está organizado sequencialmente, pois se encontra dividido nos tratadores de cada tipo de mensagem. Contudo, o programador poderá diagramar as máquinas de estado de cada nó do sistema, afim de simplificar a modelagem das interações remotas e facilitar os testes.

Em relação aos obstáculos na conectividade da rede (Requisito 6 - Conectividade

parcial), os MOMs podem tirar vantagem de sua arquitetura. As mensagens sempre passam por uma entidade intermediária, que pode ser usada 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* para permitir que todas as aplicações em um domínio se comuniquem com outros domínios.

Os MOMs permitem que as mensagens sejam armazenadas pela plataforma quando um componente estiver indisponível (Requisito 1 - Detecção de falha dos nós), de modo que sejam repassadas para ele quando se tornar disponível. Todavia, alguns sistemas não podem agir desse modo, pois precisam reagir imediatamente após a falha de algum componente.

O XMPP – *Extensible Messaging and Presence Protocol* [38] foi escolhido para participar da avaliação em relação ao atendimento dos requisitos não-funcionais definidos na seção 1.3.1, por ser o protocolo para MOMs que tem se tornado o padrão *de facto* para comunicação assíncrona.

2.2.1 XMPP

O XMPP é um exemplo de MOM, com tecnologia aberta, que pode ser utilizado em aplicações de mensagens instantâneas, presença, *group chat*, video conferência e colaboração. Esse protocolo evoluiu, se tornando um padrão para troca de mensagens assíncronas.

Existem dois elementos principais no protocolo XMPP, o cliente e o servidor. Os clientes trocam mensagens entre si, roteando-as através dos servidores. Cada cliente possui uma conta em um servidor, de modo que o endereçamento dos clientes possui o formato `login-cliente@endereço-servidor`, semelhante ao do serviço de e-mails. A troca de mensagens entre dois clientes XMPP pode trafegar por no máximo dois servidores, o que representa o cenário onde os dois clientes pertencem a servidores distintos.

Na abordagem dos objetos distribuídos, é necessário que a configuração da rede permita o acesso direto entre os nós dos sistemas, portanto não pode haver obstáculos como *firewall* na comunicação. Ao se utilizar MOMs, estes obstáculos podem ser ultrapassados facilmente, pois o acesso direto é necessário apenas entre os servidores. Logo, os clientes podem estar sob *firewall* ou rede privada.

A fim de ilustrar a facilidade da configuração da rede para o uso de MOMs, vamos mostrar o cenário específico do XMPP, que utiliza a porta 5269 para a comunicação entre os

servidores e as portas 5222 e 5223 para a comunicação servidor – cliente. Assim sendo, um site XMPP pode ter seu *firewall* organizado de duas formas:

- Com o servidor XMPP protegido pelo *firewall* - portanto deverá ser configurada a seguinte regra no *firewall*:
  - Permitir as conexões de saída e entrada, do mundo para o servidor XMPP, na porta 5269;
- Com o servidor XMPP fora do *firewall* - logo deverá ser configurada a seguinte regra no *firewall*:
  - Permitir as conexões de saída e entrada, do servidor XMPP para todos os clientes XMPP, nas portas 5222 e 5223;

Em ambos os casos, os clientes não podem ser acessados diretamente por elementos estranhos e o *firewall* pode ser facilmente configurado, necessitando de uma regra apenas.

O protocolo XMPP define uma porta segura (5223) para comunicação cliente – servidor. Além disso, os servidores podem ser configurados para trabalhar com certificados X.509. Portanto, esse protocolo atende ao Requisito 5 - Segurança. Quanto à detecção de falhas no XMPP, esta é implementada através de um protocolo de presença que indica se um nó está disponível naquele instante. Dessa forma, o XMPP suporta falhas por parada dos nós (Requisito 1 - Detecção de falha dos nós), mas não as falhas de omissão decorrentes da perda de mensagens em conexões que envolvam mais de um servidor XMPP (Requisito 2 - Detecção de perda de mensagens).

2.3 Abordagem híbrida

Para o desenvolvimento de sistemas distribuídos que precisam manter muitas conexões abertas simultaneamente em algum nó, o ideal é que a plataforma de comunicação combine as abordagens síncrona e assíncrona. A interface orientada a objetos, do modelo de objetos distribuídos (síncrono) contribui para a modularização do sistema e o assincronismo dos MOMs evita a explosão do número de fluxos de execução, melhorando a escalabilidade do sistema e, principalmente, facilitando a tarefa de testar o sistema. Escolhemos a especificação de

CORBA assíncrono e o sistema de **middleware** *JIC - Java Internet Communication* para avaliar a abordagem híbrida.

2.3.1 CORBA assíncrono

Segundo Gokhale e Schmidt [15], inicialmente a especificação **CORBA** definiu três modelos de invocação, todos síncronos: *Twoway synchronous*, *Oneway synchronous* e *Deferred synchronous*. Posteriormente, a versão 2.4 de CORBA incluiu um capítulo sobre o serviço de mensagens [26; 27], que possibilita a invocação de métodos remotos assincronamente. Esse serviço é chamado de *AMI - Asynchronous Method Invocation* e possui duas abordagens: no *polling model*, cada invocação remota retorna para o cliente um objeto do tipo *Poller*, que pode ser utilizado, a qualquer momento, para consultar o estado da invocação. Após o servidor concluir a computação remota, o objeto *Poller* também poderá informar o resultado da invocação; no *callback model*, o cliente define uma referência para um objeto de *callback*, antes de uma invocação remota. Quando o servidor terminar a computação, esse objeto de *callback* é invocado pelo sistema de *middleware*. O modelo com *callback* é mais eficiente, pois o cliente não precisa ficar consultando o objeto *Poller* periodicamente. Todavia, ele obriga o cliente a atuar como um servidor, o que pode tornar o software do cliente mais complexo.

Embora CORBA assíncrono possua uma boa arquitetura para o nicho de sistemas estudado nesta dissertação, essa especificação possui as mesmas lacunas do CORBA síncrono (ver seção 2.1.1), quais sejam, lacunas de segurança (Requisito 5), necessidade de abrir portas no *firewall* (Requisito 6 - Conectividade parcial).

2.3.2 JIC - Java Internet Communication

Lima e colegas [24; 23] desenvolveram o *JIC - Java Internet Communication*, um sistema de *middleware* com abordagem híbrida para o desenvolvimento de sistemas distribuídos P2P que precisam manter múltiplas conexões abertas.

O JIC mescla uma API orientada a objetos com a abordagem assíncrona. Isto é possível porque no JIC os métodos dos objetos remotos não possuem tipo de retorno e não lançam exceções. Portanto, após o envio das mensagens remotas, o fluxo de execução do emissor

(cliente) pode ser liberado para continuar sua execução, enquanto a computação remota está sendo executada no receptor da mensagem (servidor). Se o cliente espera algum resultado da execução remota, ele deve enviar uma referência de si mesmo para o servidor na invocação do método remoto. Essa referência é chamada de *callback* e é utilizada pelo servidor para enviar mensagens para o cliente.

A comunicação remota entre os nós do sistema é suportada por um serviço de roteamento de mensagens que utiliza servidores intermediários, semelhante ao que acontece no serviço de e-mail, permitindo a operação em ambientes com conectividade parcial. O JIC também define um mecanismo de detecção e tratamento de falhas, integrado na API orientada a objetos, que facilita o encapsulamento e o entendimento do código de tratamento de falhas do sistema.

O JIC foi utilizado em alguns projetos de código aberto, como por exemplo, a plataforma para grade computacional OurGrid [7] e o serviço de informações para grade computacional NodeWiz [3]. Até a versão 3, o OurGrid utilizava Java RMI para comunicação entre os componentes da grade computacional. A partir da versão 4, o OurGrid passou a utilizar o JIC e o seu código foi simplificado. Houve uma redução na quantidade de blocos sincronizados, de 179 para 66, pois a comunicação remota deixou de ser bloqueante e na maior parte do código havia apenas um fluxo de execução da aplicação ativo. Experimentos mostram que o JIC tem desempenho comparável a Java RMI [23].

Embora o trabalho de Lima et al. defina formalmente a semântica de falha suportada pelo sistema de *middleware*, ele não detalha como essa semântica poderia ser implementada. Por esse motivo, a implementação do JIC é capaz de detectar apenas as falhas por parada dos nós e a detecção de falhas de omissão decorrentes da perda de mensagens não pôde ser implementada efetivamente. Além disso, o JIC não oferece qualquer mecanismo de segurança para proteger os sistemas de ataques realizados por nós maliciosos.

Analisando as características do JIC, verificamos que esse sistema de *middleware* atende a cinco dos requisitos não funcionais elicitados na seção 1.3.1, pois o JIC é capaz de detectar falhas por parada dos nós (Req. 1), implementa um protocolo de comunicação robusto (Req. 3), possui uma API que facilita a comunicação remota e a implementação de testes (Req. 4), utiliza uma arquitetura com servidores intermediários que viabiliza a implantação de sistemas em ambientes com conectividade parcial (Req. 6) e provê uma API bem integrada

a linguagem orientada a objetos Java (Req. 7). Consequentemente, as lacunas do JIC são os requisitos *2 - Detecção de perda de mensagens* e *5 - Segurança*.

Existe um protoloco para sistemas distribuídos na escala da Internet chamado *Linked Process*, que utiliza XMPP para implementar a comunicação de objetos distribuídos. O sistema de *middleware LoPSideD*[2] é uma implementação em Java do Linked Process, porém decidimos não incluí-lo na avaliação dos trabalhos relacionados, por ainda ser um protótipo e pelo fato de ser uma solução específica para computação na nuvem, com clientes que são consumidores de recursos e máquinas virtuais que doam os recursos dos servidores.

2.4 Soluções para outras linguagens de programação

Realizamos uma pesquisa em busca de sistemas de *middleware* desenvolvidos em outras linguagens que também atendessem aos requisitos funcionais listados na seção 1.3.1. Encontramos o *framework Twisted* [13] construído na linguagem Python com um modelo de programação orientado a eventos, que permite comunicação através de uma enorme gama de protocolos, como TCP, UDP, SSH e XMPP.

Dentre várias formas de programação, Twisted permite a comunicação assíncrona bem integrada à linguagem de programação, sem precisar lidar com a complexidade do modelo de *Threads*. O componente principal da arquitetura *event-driven* do Twisted é o *event-loop*. 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, essa função retorna imediatamente um objeto do tipo *Deferred*. No Twisted, é permitido adicionar funções como *callback* em um *deferred*. Dessa forma, após o resultado da invocação remota, o método de *callback* associado ao *deferred* dessa invocação será executado no lado cliente.

O Twisted pode usar os *timeouts* da camada de comunicação para detectar falhas de componentes e repassar tais falhas através de exceções, usando os *deferred*. Em relação à comunicação na presença de *firewalls* e NATs, Twisted também é dependente do protocolo

[2]http://code.google.com/p/linkedprocess/

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, se o protocolo escolhido for o XMPP, o Twisted poderá se beneficiar das facilidades para administração de *firewalls*.

A avaliação do Twisted não entrará no quadro comparativo da próxima seção, pois o escopo do nosso trabalho se limita à linguagem Java.

2.5 Quadro comparativo

A Tabela 2.1 apresenta um quadro comparativo, que resume a avaliação das tecnologias estudadas neste Capítulo em relação ao atendimento dos requisitos elicitados na seção 1.3.1.

Pode-se verificar que nenhuma das tecnologias avaliadas atende a todos os requisitos. XMPP foi um dos sistemas de *middleware* melhor avaliados, atendendo a cinco requisitos não-funcionais. No entanto, essa tecnologia possui lacunas na detecção de perda de mensagens e na integração com linguagens orientadas a objeto.

Diante do exposto, concluímos que nenhuma das tecnologias avaliadas suporta completamente o desenvolvimento de sistemas P2P em Java, cujos nós precisam manter muitas conexões simultaneamente. Portanto, propomos o desenvolvimento de um novo sistema de *middleware* para esse nicho de sistemas distribuídos. No Capítulo 3, mostraremos os detalhes da solução proposta.

	Requisitos não-funcionais	Tecnologias				
		CORBA síncrono	Java RMI	XMPP	CORBA assíncrono	JIC
1.	Detecção de indisponibilidade	sim	sim	sim	sim	sim
2.	Detecção de perda de mensagens	sim	sim	não	sim	não
3.	Concorrência no middleware	sim	sim	sim	sim	sim
4.	Facilidade de programação	não	não	sim	sim	sim
5.	Segurança	parcial	parcial	sim	parcial	não
6.	Conectividade parcial	não	não	sim	não	sim
7.	Orientação a objetos	sim	sim	não	sim	sim

Tabela 2.1: Quadro comparativo sobre tecnologias para sistemas distribuídos sobre a Internet

Capítulo 3

O sistema de middleware Commune

O Capítulo 3 descreve a solução proposta por este trabalho de dissertação, um novo sistema de *middleware* que atende aos requisitos não-funcionais de sistemas P2P em Java, cujos nós precisam manter muitas conexões simultâneas. Inicialmente, apresentaremos a arquitetura da nossa solução, chamada *Commune*. Posteriormente, descreveremos as suas duas principais contribuições, os mecanismos de segurança e um serviço para a detecção de falhas em conexões capaz de detectar a perda de mensagens no meio de comunicação.

3.1 Arquitetura

Considerando as mesmas ideias e motivações do JIC, mas resolvendo as lacunas de segurança e detecção de falhas, desenvolvemos um novo sistema de *middleware*, denominado *Commune*. Desse modo, todos os sete requisitos descritos na seção 1.3.1 foram atendidos e o Commune mostrou-se ser uma boa opção para o desenvolvimento de sistemas distribuídos P2P em Java cujos nós requerem a manutenção de muitas conexões entre si.

Na arquitetura do Commune, um sistema distribuído é composto por um conjunto de nós que se comunicam através da Internet. Cada nó representa uma parte da aplicação e executa dentro de uma máquina virtual Java. A Figura 3.1 descreve a arquitetura geral de um nó Commune.

De acordo com a arquitetuta apresentada, o nó de um sistema distribuído feito sobre o Commune é composto por: (i) o software da Aplicação que contém a lógica de negócio e é desenvolvida pelo usuário, sendo representada pelo componente *Application*; e (ii) o nó

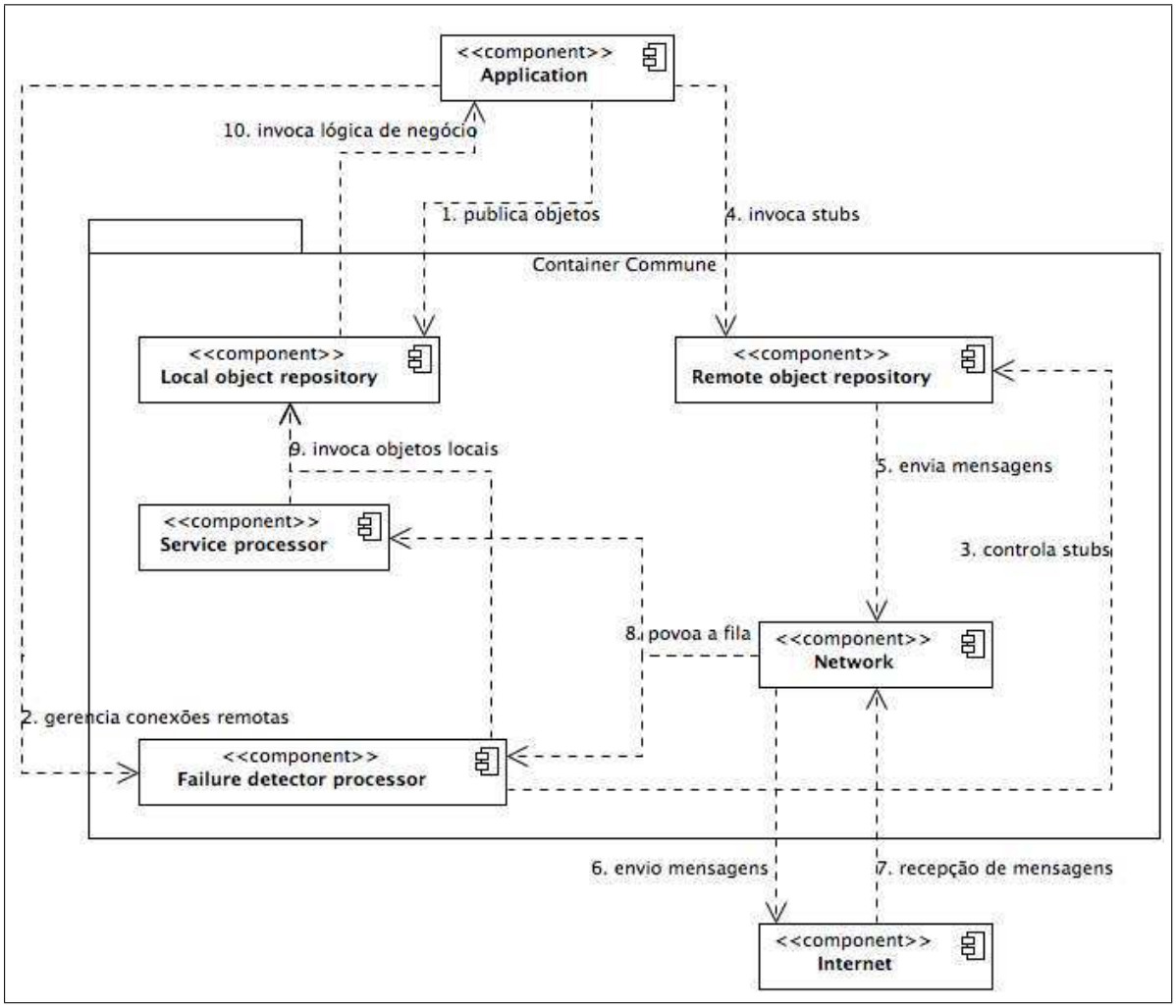

Figura 3.1: Arquitetura de um nó Commune

Commune, que viabiliza a comunicação com outros nós através da Internet.

O nó Commune atua como um *Container* onde são publicados objetos remotos. Uma vez publicados, os métodos desses objetos podem ser invocados remotamente por outros nós Commune, através de mensagens que trafegam pela Internet. Assim sendo, cada nó Commune possui dois repositórios: o *Local object repository* armazena os objetos locais publicados pela aplicação e representa o lado servidor na invocação remota; e o *Remote object repository*, que contém os *stubs*[1] para os objetos publicados em nós remotos, representa o lado cliente. Toda a comunicação remota do Commune trafega pelo componente *Network*, que faz a interface do nó com a Internet, para o envio e recebimento de mensagens. Além disso, no Container Commune existem dois processadores, cada um possui: (i) uma

[1]Um *stub* é um objeto com interface idêntica ao objeto remoto e que serve de *proxy* para este.

fila de mensagens que é povoada pelo *Network* e (ii) um fluxo de execução que consome as mensagens e atua sobre os repositórios de objetos e de *stubs*.

A interação entre os elementos da arquitetura Commune inicia quando a aplicação de algum nó (componente *Application*) decide atuar como servidor e publica algum objeto no *Local object repository*, ver mensagem (1) na Figura 3.1.

Em outro nó, a aplicação cliente decide abrir uma conexão com o servidor, mandando uma mensagem (2) para o *Failure detector processor* se interessar pelo servidor. Daí em diante, o serviço de detecção de falhas começa a monitorar o servidor.

Quando a presença do servidor é detectada, o componente *Failure detector processor* do cliente cria um *stub* no *Remote object repository* (3) e invoca um dos objetos locais (9) para notificar a ocorrência da recuperação do servidor.

Nesse instante, a conexão no sentido cliente -> servidor acabou de ser estabelecida e a aplicação do cliente poderá invocar o *stub* do servidor (4). Essa invocação é convertida em uma mensagem Commune, que é enviada pelo componente *Network* (5) para a Internet (6).

No lado servidor, a Internet repassa a mensagem para o componente *Network* (7), que a roteia para algum processador de acordo com o tipo da mensagem (8). Como nesse exemplo estamos tratando de uma mensagem de aplicação, a mensagem Commune será roteada para a fila do *Service processor*. Depois disto, quando a mensagem for consumida, o *Service processor* invoca no objeto local o mesmo método que foi invocado no *stub* pelo cliente, com os mesmos parâmetros (9). Por fim, o objeto local repassa a execução para a lógica da aplicação servidora (10).

Mesmo após a recuperação de um objeto remoto, o componente *Failure detector processor* continua monitorando a sua disponibilidade. Se ocorrer alguma falha na conexão, a aplicação receberá uma notificação de falha. A monitoração será encerrada e o *stub* será removido quando a aplicação decidir liberar a conexão remota, avisando ao *Failure detector processor*.

Nas próximas seções, detalharemos a principais contribuições do Commune, que resolveram as lacunas do JIC com relação a segurança e detecção de perda de mensagens. Essas contribuições permitiram que o Commune se tornasse um sistema de *middleware* bem adaptado para o nicho de sistemas distribuídos estudado neste trabalho.

3.2 Resolvendo segurança

Utilizamos o guia de segurança da OMG [12] como fonte para o levantamento dos requisitos de segurança para sistemas de objetos distribuídos. Os principais tópicos de segurança estão relacionados com:

Confidencialidade: as informações só podem ser acessadas por pessoas autorizadas;

Integridade: as informações só podem ser modificadas por pessoas autorizadas;

Auditoria: as ações dos usuários podem ser auditadas, quando elas forem relevantes nos aspectos de segurança;

Disponibilidade: o uso do sistema não pode ser negado maliciosamente para os usuários autorizados.

No contexto dos sistemas distribuídos, a segurança se torna ainda mais importante, pois a informação em trânsito é mais vulnerável. Além disso, será necessário tratar questões de confiança e consistência entre os diversos componentes remotos do sistema. A seguir apresentamos um conjunto de requisitos gerais para o gerenciamento da segurança de sistemas distribuídos e heterogêneos.

Consistência. O modelo de segurança deve oferecer uma abordagem consistente e padronizada para facilitar a transportabilidade entre plataformas diferentes. Isto inclui uso de políticas comuns para o acesso a informações dentro de um domínio composto por máquinas heterogêneas; compatibilidade com os mecanismos de permissão existentes; provisão de segurança fim-a-fim mesmo que os meios de comunicação sejam inseguros; reaproveitamento dos mecanismos de *logon* e com os bancos de dados existentes, a fim de não demandar *logons* adicionais nem administração adicional de usuários.

Escalabilidade. Deve ser possível prover segurança tanto para sistemas pequenos como para sistemas grandes. Para estes, será necessário manipular as permissões de grupos em vez de indivíduos, para reduzir os custos administrativos. Além disto, deve permitir a criação de domínios de segurança, cada um com suas políticas, mas interoperáveis entre si. A distribuição de chaves criptográficas não deve impor mais esforços administrativos, portanto sugere-se o uso da tecnologia de chaves públicas.

Transportabilidade. As interfaces para os serviços de segurança, como por exemplo, autenticação, devem ser independentes do protocolo utilizado, de modo que os mesmos objetos poderão ser utilizados em diferentes serviços e mecanismos de segurança.

Usabilidade. Se a segurança for difícil de administrar e usar, os usuários criarão alternativas para não utilizá-la, portanto o sistema se tornará inseguro ou o usuários simplesmente o abandonarão. O modelo de segurança deve ser usável para os usuários, administradores e programadores.

Desempenho. A segurança não deve impor uma sobrecarga inaceitável no desempenho. Particularmente, para os níveis de segurança de aplicações comerciais. Se houver níveis de segurança maiores, uma sobrecarga maior será aceitável.

Flexibilidade. Como as políticas de segurança variam de instituição para instituição, deve ser possível escolher os mecanismos de segurança a serem utilizados. Isto permite que se pague pelo custo de segurança apenas quando o nível de proteção é necessário. O nível de segurança poderá ser reduzido quando a informação for menos sensível ou quando o sistema é menos vulnerável a ataques.

Interoperabilidade. O modelo de segurança deve permitir a interoperabilidade de objetos mesmo que (i) utilizem protocolos e serviços de segurança distintos, (ii) estejam sob políticas de segurança diferentes e (iii) os objetos remotos pertençam a sistemas inseguros.

A escolha dos mecanismos de segurança introduzidos no Commune foi feita mediante o estudo prévio sobre as principais opções disponíveis na literatura. Como resultado desse estudo, decidimos implementar dois níveis de segurança no Commune.

No primeiro nível de segurança, o sistema é capaz de resistir a ataques de *spoofing*, que podem ocorrer quando o endereço IP ou o domínio do remetente de uma mensagem é forjado por um nó malicioso. Neste caso, o nó malicioso pode enviar mensagens se passando por um remetente válido, de modo que a comunicação entre dois nós legítimos pode se tornar inconsistente ou, no pior dos casos, que algum acesso a informação sensível seja liberado para o nó malicioso. Esse problema pode ocorrer nos sistemas que utilizam o endereço do remetente de mensagens como um identificador na comunicação remota. Portanto, é necessário prover um identificador único e seguro para cada nó, a fim de (i) montar um histórico persistente da comunicação entre os nós e (ii) se proteger dos ataques de *spoofing*.

No segundo nível de segurança do Commune, além de um identificador único e seguro, os nós precisarão associá-lo a uma identidade conhecida, como um nome ou um domínio. Assim sendo, as políticas de segurança poderão ser configuradas utilizando identificadores legíveis para os nós do sistema.

Existem diversos serviços que atendem ao primeiro nível de segurança, entretanto, alguns requerem o uso de servidores, como o Kerberos [21], ou mesmo a configuração e uso de uma autoridade certificadora para controlar a troca de identificações no sistema [1]. Escolhemos, portanto, uma solução mais simples, próxima ao SPKI [11]. Cada nó Commune contém um par de chaves assimétricas [22]: chave privada e chave pública. A chave privada é secreta e é usada para assinar cada mensagem enviada pelo nó, enquanto a chave pública é anexada às mensagens enviadas, sendo assim conhecida por todos os seus nós receptores. Se a chave pública está correta, ela pode ser usada para validar a assinatura das mensagens, garantindo ao nó receptor que a chave pública pertence ao nó remetente. Portanto, a chave pública pode ser utilizada como um identificador único do nó emissor da mensagem. Além disso, essa solução é usada pelo serviços internos do Commune para evitar que um nó malicioso possa forjar o endereço do remetente ou promover ataques de negação de serviço.

A solução utilizada para implementar o segundo nível de segurança é baseada nos certificados X.509 validados por autoridades certificadoras [1]. O Commune anexa o certificado do emissor nas mensagens trocada entre dois nós. A aplicação do nó receptor pode definir em quais autoridades confia. Assim, se o nó remetente tem um certificado que é assinado por qualquer uma dessas autoridades certificadoras, então o nó receptor pode confiar no nome ou domínio do nó remetente. A aplicação que utiliza a certificação pode implementar políticas de funcionamento e segurança baseadas na identidade dos nós remetentes (remotos).

A implantação de sistemas distribuídos que usam o sistema de *middleware* Commune procede da seguinte forma. Por padrão, um nó Commune gera automaticamente o seu par de chaves assimétricas e um certificado auto-assinado. Se o par de chaves ou o certificado são importantes para a aplicação, esses dados gerados automaticamente podem ser substituídos por outros, por exemplo, obtidos de uma autoridade certificadora. Caso contrário, a aplicação apenas ignora as chaves e o certificado gerados, sendo esses usados apenas nos componentes internos do Commune, para garantir que a troca de mensagens se faz de modo seguro, ou seja, sem a possibilidade de mensagens serem forjadas por nós maliciosos.

A solução de segurança está localizada no componente *Network* da arquitetura do Commune. Quando as mensagens são enviadas e recebidas, os componentes de assinatura de mensagens e de certificação atuam, alterando os dados de segurança das mensagens e eliminando as mensagens inválidas. Essa solução é flexível e independente do protocolo de comunicação utilizado.

Ao comparar a implementação dos mecanismos de segurança do Commune com os requisitos do guia de segurança da OMG [12], podemos verificar que o Commune provê o suporte para o atendimento da maior parte dos requisitos, conforme visto a seguir.

O mecanismo de assinatura das mensagens com chaves assimétricas não permite que as mensagens sejam alteradas por nós maliciosos (integridade), nem que as sessões de comunicação entre dois nós sejam maliciosamente invalidadas (disponibilidade).

Em relação ao desempenho, a escolha dos algoritmos de assinatura de mensagens levou em consideração os algoritmos que combinavam da melhor forma as características de segurança e desempenho (ver os detalhes no Capítulo 4). Na seção 5.3, apresentaremos os resultados de alguns experimentos realizados para medir a sobrecarga da troca de mensagens do Commune em relação a Java RMI. Verificamos que a sobrecarga adicionada não se mostrou proibitiva.

O Commune utiliza uma biblioteca para fazer *log* das mensagens que são enviadas e recebidas por cada nó. Através desse mecanismo, pode-se gravar em arquivo ou banco de dados o certificado correspondente a qualquer operação no sistema distribuído, viabilizando uma posterior auditoria.

A escalabilidade dos sistemas distribuídos feitos sobre o Commune é facilitada por causa da certificação e das chaves públicas. A certificação permite aos administradores definirem políticas na granularidade de grupos em vez de individualmente. Esses grupos são definidos de acordo com a autoridade certificadora que validou o certificado de cada nó. Por exemplo, para os nós validados pela autoridade certificadora X, aplicar a política A. No entanto, nada impede que a aplicação possa utilizar outros serviços para definir os grupos de acordo com outros critérios, como é feito no *Virtual Organization Membership Service - VOMS* [10], por exemplo. A tecnologia de chaves públicas também aumenta a escalabilidade do sistema, pois não será necessária a intervenção humana para que as chaves sejam distribuídas pelo sistema.

Os componentes de segurança atuam sobre as mensagens do Commune independentemente do protocolo de comunicação utilizado, atendendo, desse modo, ao requisito de transportabilidade.

Os requisitos de usabilidade, flexibilidade e interoperabilidade são atendidos pela geração automática de pares de chaves assimétricas e de certificados. Isto torna a administração e o uso da segurança mais fácil e usável, além de permitir que o nível de segurança seja flexível. Como todos os nós possuem chaves e certificados *default*, domínios com políticas de segurança diferentes podem interoperar, como por exemplo, *sites* seguros podem se comunicar com outros inseguros.

Em relação ao requisito de consistência, o Commune é beneficiado pelo fato de ter sido construído sobre a plataforma Java, o que lhe confere a capacidade de ser implantado em diversos tipos de plataformas de hardware e software diferentes. Devido à disseminação das tecnologias de autenticação e certificação escolhidas para o Commune, é possível manter a compatibilidade com a maioria dos mecanismos de permissão existentes. O Commune também é capaz de implementar a segurança fim-a-fim, pois seus componentes de segurança são independentes do protocolo de comunicação utilizado. No entanto, o Commune pode exigir esforço extra de segurança, caso os mecanismos legados de segurança não utilizem chaves assimétricas ou certificação.

O requisito de confidencialidade ainda não é atendido pelo Commune, mas o algoritmo de assinatura de mensagens escolhido (ver Capítulo 4) permite que a aplicação implemente canais confiáveis.

3.3 Suporte ao tratamento de falhas de conexão

A outra contribuição do Commune foi detalhar como poderia ser implementada uma semântica para detecção de falhas de conexão capaz de detectar a perda de mensagens no meio de comunicação. Nesta seção, revisaremos o modelo de conexão e a semântica de detecção de falhas definidos pelo JIC. Posteriormente, detalharemos como esse modelo foi estendido no Commune para ser capaz de detectar a perda de mensagens no meio de comunicação.

3.3.1 Modelo de conexão do JIC

Em cada nó N do JIC, existe um componente de detecção de falhas D_N. Quando um nó A precisa enviar uma mensagem para um nó remoto B, A registra o interesse por B no seu componente de detecção de falhas D_A. Daí em diante, D_A cria uma conexão para B, cujo estado é *indisponível*, e tenta se comunicar com D_B a fim de verificar a presença de B. O serviço de detecção de falhas do JIC utiliza o modelo de monitoramento *pull* [23], cujo algoritmo pode ser entendido da seguinte forma:

- Periodicamente, D_A envia para D_B a mensagem *"B está vivo?"*
- Sempre que B estiver disponível, D_B responde para D_A com a mensagem *"B está vivo!"*
- Ao receber a primeira resposta, D_A marca a conexão como *disponível* e notifica a recuperação de B para a aplicação em A.
- Após a recuperação, D_A continua enviando mensagens periodicamente para D_B.
- Se as mensagens não forem respondidas em um determinado período de tempo (*timeout*), D_A marca a conexão como *indisponível* e notifica a falha de B para a aplicação em A.
- Neste ponto, o algoritmo é reiniciado em D_A, que continuará enviando mensagens *"B está vivo?"*
- O monitoramento e conexão só serão encerrados quando a aplicação em A desregistrar o interesse por B em D_A.

A Figura 3.2 explica os elementos e os dados de uma conexão entre dois nós no JIC. Entre os nós A e B, é criada uma conexão composta por três canais para o tráfego de mensagens. Os dois canais de controle são utilizados para trafegar as mensagens entre D_A e D_B nos dois sentidos. Existe apenas um canal de dados que, neste caso, é utilizado por A para enviar mensagens de aplicação para B. Conceitualmente, o sentido do canal de dados define o sentido da conexão como um todo.

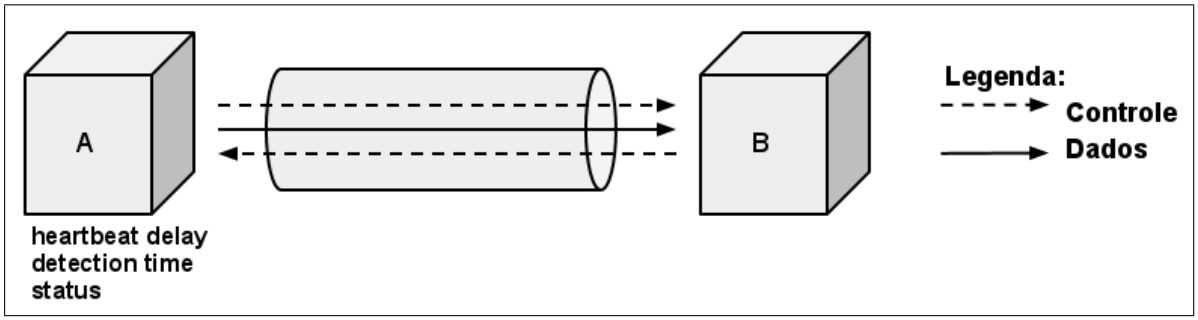

Figura 3.2: Conexão JIC

Dentro do nó cliente de uma conexão (A), existe um mapa de conexões com nós remotos. Para cada conexão são guardadas as informações sobre periodicidade em milissegundos do envio das mensagens *"está vivo?"* – *heartbeat delay*; tempo máximo sem receber a mensagem de disponibilidade de B (*"está vivo!"*) até suspeitar da sua falha – *detection time*; e o estado do nó servidor da conexão (disponível ou indisponível) – *status*.

É interessante observar que geralmente, quando A envia mensagens de aplicação para B, B também precisará responder com mensagens de aplicação para A. Neste cenário, haverá duas conexões com estados independentes, uma de A para B e outra de B para A. Assim sendo, pode-se dizer que a *comunicação* entre dois nós envolve até duas conexões.

Uma conexão JIC entre dois nós, A e B, pode requerer o roteamento de mensagens utilizando até três conexões TCP. Neste caso, as mensagens de aplicação podem ser perdidas por causa do roteamento. O modelo de conexões JIC não é capaz de detectar a perda de mensagens de aplicação no protocolo de mais baixo nível, pois o único mecanismo de suspeita de falhas é o *detection time*. Por exemplo, se após a perda de uma mensagem de aplicação de A para B, D_A recebe uma mensagem *"está vivo!"* de D_B, então o *detection time* será reiniciado em A e a falha não será notada.

3.3.2 Semântica de detecção de falhas do Commune

Para atender ao requisito *2 - Detecção de perda de mensagens*, o Commune precisou aprimorar o mecanismo de detecção de falhas do JIC. Foi adicionado um protocolo de conexão que possui uma semântica de detecção de falhas clara e abrangente, cobrindo tanto falhas por parada dos nós como falhas de omissão decorrentes da perda de mensagens.

A seguir definiremos os predicados que esse protocolo de conexão deve atender. No

entanto, explicaremos antes o significado dos operadores temporais e de alguns símbolos utilizados nesses predicados.

Os predicados do protocolo de conexão utilizam conceitos temporais para definir a semântica de detecção de falhas. Portanto, apresentaremos alguns operadores da lógica temporal que serão usados [41]:

- $F(p)$ - em algum momento do futuro a proposição p será verdade;
- $G(p)$ - no futuro a proposição p sempre será verdade;
- $P(p)$ - em algum momento do passado a proposição p foi verdade.

Utilizaremos as letras A e B para representar os nós presentes no cenário dos predicados. Se a letra for maiúscula, significa que o nó está disponível ou que o seu estado na conexão é disponível. Por outro lado, se a letra estiver minúscula, o nó está indisponível. Além disso, definiremos alguns operadores binários para representar o estado de uma conexão entre dois nós:

1. *A » B* - indica que A está interessado em B, independente da disponibilidade de B;
2. *A $\not$> b* - significa que A está interessado em B, mas A está suspeitando que B está indisponível, por isto na expressão a letra b está minúscula;
3. *A −> B* - representa uma conexão estabelecida, onde B está disponível e mensagens de aplicação podem ser enviadas de A para B;
4. *A $\not$> B* - significa que houve uma perda de mensagem no meio de comunicação. A diferença para o segundo operador está no fato do segundo operando estar maiúsculo.

Por fim, também serão utilizados os operadores básicos "e", "ou", "!"(negação) e "Se ..., então ...".

Predicados para o protocolo de conexão do Commune

Com o objetivo de definir uma semântica de detecção de falhas abrangente, capaz de detectar as falhas por parada dos nós e as falhas de omissão decorrentes da perda de mensagens, o protocolo de conexão do Commune deve atender aos quatro predicados a seguir.

Predicado 1: Estabelecimento e manutenção da conexão

Se *G(A e B e A » B e !(A +> B))*, então *F(A −> B)*

Se *(A −> B e G(A) e G(B) e !F(A +> B))*, então *G(A −> B)*

A primeira parte do predicados diz que, se A e B sempre estarão disponíveis, A sempre estará interessado por B e não haverá perda de mensagens de aplicação de A para B, então é certo que em algum momento A conseguirá estabelecer uma conexão com B.

A segunda parte significa que, se A está conectado a B, sempre A e B estarão disponíveis e nunca haverá perda de mensagens na conexão de A para B, então A permanecerá conectado para sempre com B.

Predicado 2: Detecção de falha por parada dos nós

Se *(A −> B e F(b))*, então *F(A +> b)*

Se A está conectado a B e B se tornar indisponível, então em algum momento A suspeitará a falha de B e invalidará a conexão.

Predicado 3: Detecção de falha por omissão devido a perda de mensagem

Se *(A −> B e F(A +> B))*, então *F(A +> b)*

Se *(A −> B e B −> A e (F(A +> B) ou F(B +> A)))*, então *F(A +> b) e F(B +> a)*

A primeira parte do predicados trata da perda de mensagens, quando apenas um nó está enviando mensagens de aplicação para o outro. Se A está conectado a B e ocorrer uma perda de mensagem no canal de A para B, então em algum momento A detectará a falha de B e invalidará a conexão.

Na segunda parte, a troca de mensagens de aplicação está fluindo nos dois sentidos. Se A está conectado a B, B está conectado a A e ocorrer uma perda de mensagem no canal de A para B ou no canal de B para A, então em algum momento A detectará a falha de B, B detectará a falha de A e as duas conexões serão invalidadas.

Predicado 4: Restabelecimento de conexão

Se *(P(A −> B) e A +> b e G(A e B e A » B e !(A +> B)))*, então *F(A −> B)*

> Se em algum momento A esteve conectado a B, atualmente A suspeita da falha de B, e de agora em diante A e B sempre estarão disponíveis, A sempre estará interessado em B e não haverá perda de mensagens de aplicação de A para B, então em algum momento A conseguirá restabelecer a conexão com B.

Esses predicados são simétricos, portanto não precisamos definir os predicados para as conexões partindo inicialmente de B para A, pois na prática eles são equivalentes aos predicados descritos acima.

O predicado 4 foi definido para deixar explícito que o protocolo não deve ter estados de impasse (*deadlock*). Desse modo, sempre será possível restabelecer a conexão entre dois nós. Como o predicado 1 representa o caminho natural para o estabelecimento de uma conexão, a partir de um interesse entre os nós, coube ao predicado 4 representar o cenário de estabelecimento de uma conexão onde ela já existira mas uma falha ocorreu.

3.3.3 Modelo para detecção de falhas

Agora definiremos o modelo de um protocolo de conexão que atende aos quatros predicados acima, conferindo ao Commune uma semântica de detecção de falhas clara e abrangente, capaz de detectar falhas por parada dos nós e falhas de omissão devido a perda de mensagens no meio de comunicação. Inicialmente, descreveremos os dados que compõem o *estado* da comunicação entre dois nós Commune. Posteriormente, explicaremos um diagrama de classes que detalha as interfaces de um nó Commune, com as *operações* que podem ser invocadas durante a comunicação entre dois nós.

O Commune utiliza os conceitos de nó e conexão de forma semelhante ao JIC, que foram descritos na seção 3.3.1, e adiciona o conceito de *sessão* na conexão entre os nós. Uma sessão é criada quando um nó (emissor) se interessa por outro nó (receptor), sendo representada por um número que a identifica unicamente e é gerado neste instante pelo emissor. Desta forma, as mensagens de controle que são enviadas para o nó receptor são marcadas com esse número de sessão, que também será utilizado pelas mensagens de aplicação no canal de dados. Quando o emissor detectar uma falha na conexão, a sessão sempre será alterada e possuirá um novo número único.

Além do número de sessão, as mensagens também serão marcadas com um número de

sequência, cujo valor é zerado na criação e na alteração da sessão. O número de sequência é incrementado pelo nó emissor, durante o envio de cada mensagem de aplicação.

O nó receptor mantém os números de sessão e sequência correntes da conexão. Se for recebida uma mensagem de controle, os números de sessão e de sequência devem ser iguais aos números correntes. Se for recebida uma mensagem de aplicação, apenas o número de sessão deve ser igual ao número corrente, pois o número de sequência foi incrementado no emissor e deve ser igual ao número corrente mais um.

Estamos assumindo que o protocolo de comunicação de mais baixo nível utilizado pelo Commune é FIFO, portanto não haverá desordem na entrega das mensagens. Deste modo, o nó receptor pode detectar quando for perdida alguma mensagem de aplicação. Basta verificar se o número de sequência é maior que o esperado, nas mensagens de aplicação ou controle. Além disto, o nó receptor pode concluir que o emissor criou uma nova sessão, se receber uma mensagem com número de sessão diferente do esperado. Isto provavelmente indica que o emissor da mensagem foi reiniciado ou que ele suspeitou da falha do nó receptor. Portanto, se o receptor também está interessado no emissor, então a aplicação no receptor será notificada da eventual falha do emissor.

As Figuras 3.3 e 3.4 mostram os dados envolvidos nas conexões Commune. Além dos dados presentes em uma conexão JIC, foram adicionados os números de sessão e sequência no emissor e no receptor das mensagens, como pode ser visto comparando as Figuras 3.2 e 3.3.

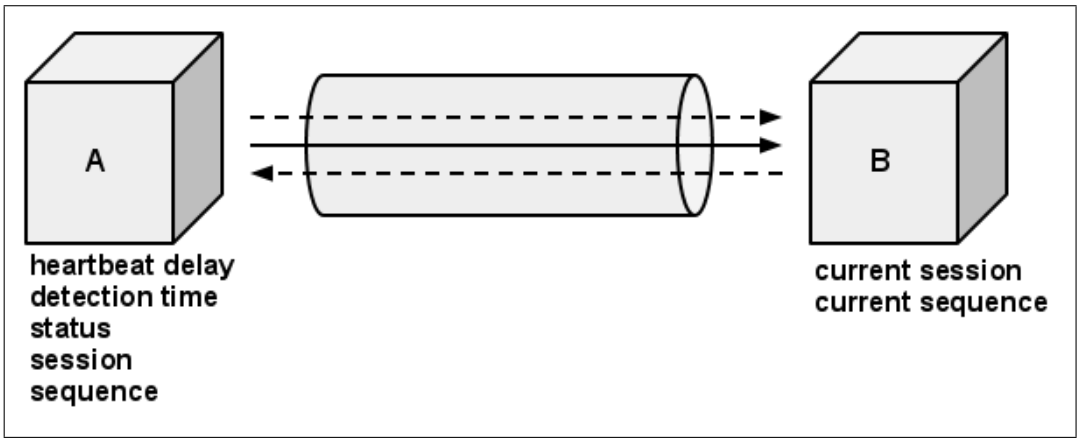

Figura 3.3: Conexão Commune

No cenário de apenas uma conexão, o nó emissor (A) trata a conexão (A –> B) como

uma conexão de saída e o nó receptor (B) trata essa conexão como uma conexão de entrada. Se o nó receptor notar alguma perda de mensagem na conexão de entrada, a sessão será invalidada e as suas mensagens serão ignoradas. Desta forma, o emissor não receberá mais as respostas para as mensagens de controle e ocorrerá um *timeout* da conexão. Daí, o emissor será notificado da falha ocorrida.

A Figura 3.4 trata do cenário de conexões reversas, onde os dois nós estão mutuamente interessados em se comunicar, conseguiram se conectar e a troca de mensagens está fluindo nos dois sentidos. Neste cenário de conexões reversas, cada nó possui uma conexão de entrada, pela qual recebe mensagens, e uma conexão de saída, utilizada para enviar mensagens. Por exemplo, para A a conexão A –> B é de saída e a conexão B –> A é a conexão de entrada.

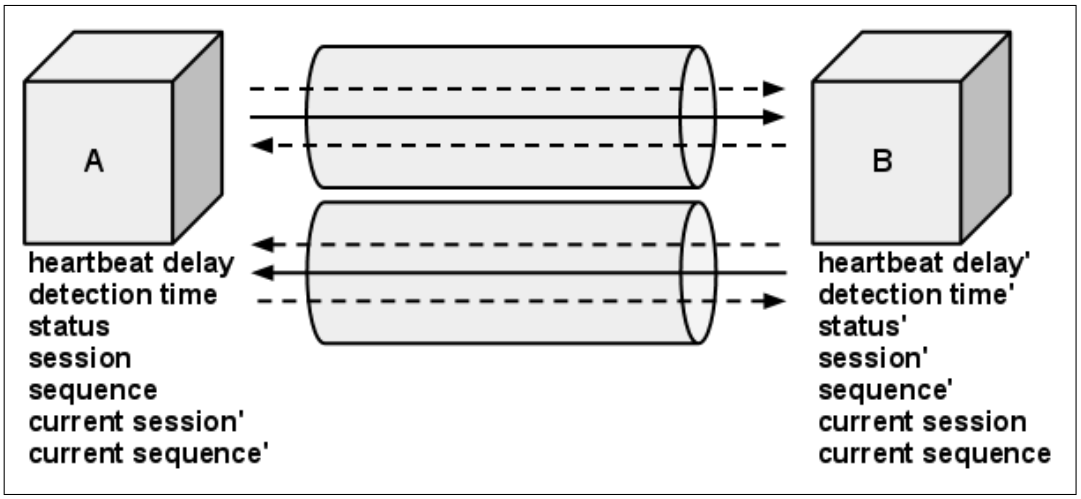

Figura 3.4: Conexões reversas no Commune

Neste caso, as duas conexões presentes são independentes e os dados da conexão são duplicados. Os números de sessão e sequência das duas conexões não são obrigatoriamente iguais. A única dependência entre essas conexões ocorre durante a suspeita de uma falha. Por exemplo, se a falha for detectada por B na conexão de entrada A –> B, a conexão de saída B –> A também será invalidada.

Trataremos agora das operações de um nó Commune que interferem no estado da comunicação entre dois nós. As interfaces presentes na Figura 3.5 agrupam essas operações de acordo com o componente que as invoca e com o propósito de suas funcionalidades. As setas pontilhadas representam trocas de mensagens assíncronas.

A interface *FailureDetectorAPI* é utilizada pela aplicação para iniciar (*registerInterest*)

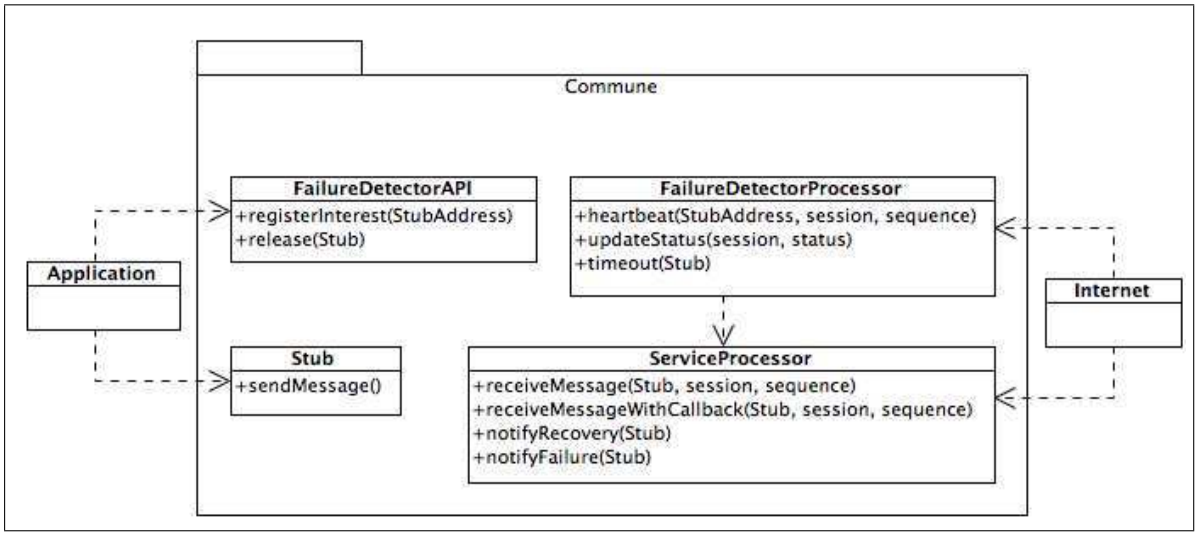

Figura 3.5: Interfaces de um nó Commune

e finalizar as conexões (*release*). Após a inicialização de uma conexão, os componentes *FailureDetectorProcessor* dos nós emissor e receptor começam a trocar mensagens de controle: o emissor pergunta se o receptor está vivo (*heartbeat*); o receptor responde informando a sua disponibilidade (*updateStatus*); e se um receptor, que atualmente está disponível, passar uma certa quantidade de tempo sem responder, um *timeout* será gerado.

FailureDetectorProcessor também é responsável por notificar a aplicação da disponibilidade e indisponibilidade do nó remoto, invocando os métodos *notifyRecovery* e *notifyFailure* do componente *ServiceProcessor*, respectivamente. Quando um nó remoto está disponível, a aplicação tem acesso a um *Stub* que pode ser invocado (*sendMessage*) para enviar uma mensagem remota. No receptor, essa mensagem é recebida no *ServiceProcessor* através do método *receiveMessage*. As mensagens específicas onde o emissor envia uma referência para si mesmo (*callback*), são recebidas pelo método *receiveMessageWithCallback*. Neste caso, o receptor cria uma conexão reversa para o emissor e assume o seu estado como disponível imediatamente, para que a referência de *callback* possa ser utilizada durante o tratamento da mensagem.

A Tabela 3.1 lista todas as operações do Commune que afetam o estado de uma conexão. Além disso, combinamos os valores possíveis para os parâmetros de cada operação como, por exemplo, número de sequência igual ao esperado ou diferente do esperado . Deste modo, a tabela resume todos os eventos que afetam as conexões Commune.

A implementação do protocolo de conexão do Commune foi feita no componente *Network*, pois as mensagens de aplicação e controle precisam ser marcadas com os números de sessão e sequência no envio; e, na recepção, esses números serão verificados antes de chegarem nos demais componentes do Commune.

O Capítulo 4 mostra os detalhes de implementação do Commune, principalmente no que tange aos mecanismos de segurança e ao protocolo de conexão.

Evento	**Parâmetros**	**Descrição**
heartbeat(ses,0)	*ses* *0*	Sessão igual à atual Sequência igual a zero, início de sessão
heartbeat(ses,seq)	*ses* *seq*	Sessão igual à atual Sequência igual à atual
heartbeat(ses,!seq)	*ses* *!seq*	Sessão igual à atual Sequência diferente à atual
heartbeat(!ses,0)	*!ses* *0*	Sessão diferente à atual Sequência igual a zero, início de sessão
heartbeat(!ses,seq)	*!ses* *seq*	Sessão diferente da atual Sequência igual à atual
heartbeat(!ses,!seq)	*!ses* *!seq*	Sessão diferente da atual Sequência diferente da atual
notifyFailure(x)	*x*	Endereço do nó remoto
notifyRecovery(X)	*X*	*Stub* do nó remoto
receiveMessage(x,ses,++seq)	*x* *ses* *++seq*	Endereço do nó remoto Sessão igual à atual Sequência igual à atual mais 1
receiveMessage(x,ses,!++seq)	*x* *ses* *!++seq*	Endereço do nó remoto Sessão igual à atual Sequência diferente da atual mais 1
receiveMessage(x,!ses,)*	*x* *!ses* *	Endereço do nó remoto Sessão diferente da atual Qualquer número de sequência
receiveMessageWithCallback (x,ses,++seq)	*x* *ses* *++seq*	Endereço do nó remoto Sessão igual à atual Sequência igual à atual mais 1
receiveMessageWithCallback(x,ses,!++seq)	*x* *ses* *!++seq*	Endereço do nó remoto Sessão igual à atual Sequência diferente da atual mais 1
receiveMessageWithCallback(x,!ses,)*	*x* *!ses* *	Endereço do nó remoto Sessão diferente da atual Qualquer número de sequência
registerInterest(x)	*x*	Endereço do nó remoto
release(X)	*X*	*Stub* do nó remoto
sendMessage()		
timeout(X)	*X*	*Stub* do nó remoto
updateStatus(ses,UP)	*ses* *UP*	Sessão igual à atual O nó remoto está disponível
updateStatus(ses,DOWN)	*ses* *DOWN*	Sessão igual à atual O nó remoto está indisponível
updateStatus(!ses,)*	*!ses* *	Sessão diferente da atual Qualquer estado

Tabela 3.1: Eventos de uma conexão Commune

Capítulo 4

Implementação

Neste Capítulo abordaremos os detalhes de implementação do Commune. Primeiramente, descreveremos quais as tecnologias adotadas para linguagem de programação, comunicação remota e segurança. Logo após abordaremos os detalhes de projeto do componente *Network*, dos mecanismos de segurança e do protocolo de conexão. Finalizaremos o Capítulo, comparando o protocolo de conexão do Commune com o TCP, e demonstrando alguns exemplos de uso do Commune.

4.1 Tecnologias utilizadas

O Commune foi implementado sobre a plataforma Java versão 5.0, utilizando o protocolo XMPP para comunicação remota. O código fonte está disponível, com login igual a *anonymous* e senha vazia, no seguinte repositório SVN:

`http://svn.lsd.ufcg.edu.br/repos/ourgrid/projects/commune`

A linguagem Java é portável para diversas plataformas, permitindo inclusive que o código fonte seja compilado em uma plataforma e que o código intermediário gerado seja executado em outra plataforma. Java é uma linguagem orientada a objetos muito popular e possui bibliotecas que oferecem suporte para a implementação dos mecanismos de segurança do Commune e facilitam a sua integração com XMPP.

No momento da escrita desta dissertação, a plataforma Java encontra-se na versão 6.0 que foi lançada há 3 anos. Essa versão foi amadurecida pelo lançamento de 18 *patches*[1]. Quando uma nova versão de Java é lançada, a linguagem assume o compromisso de manter a compatibilidade com as versões anteriores. Assim sendo, decidimos implementar o Commune utilizando a versão 5.0, pois sistemas mais antigos poderão ser compatíveis com o sistema de *middleware*. Além disso, as tecnologias de segurança e de integração com XMPP são compatíveis com o Java 5.0.

Como foi descrito na seção 2.2, o protocolo XMPP se tornou um padrão *de facto* para a comunicação assíncrona. Na sua arquitetura, XMPP define dois elementos principais, os clientes e os servidores intermediários. Os nós Commune atuam como clientes XMPP e diversos tipos de servidores XMPP poderiam ser utilizados como servidores intermediários. Atualmente, testamos o Commune apenas com o servidor Openfire[2]. A Figura 4.1 compara a representação de uma conexão Commune com uma conexão XMPP, que pode ser formada por até três conexões TCP/IP, quando os clientes pertencem a servidores distintos.

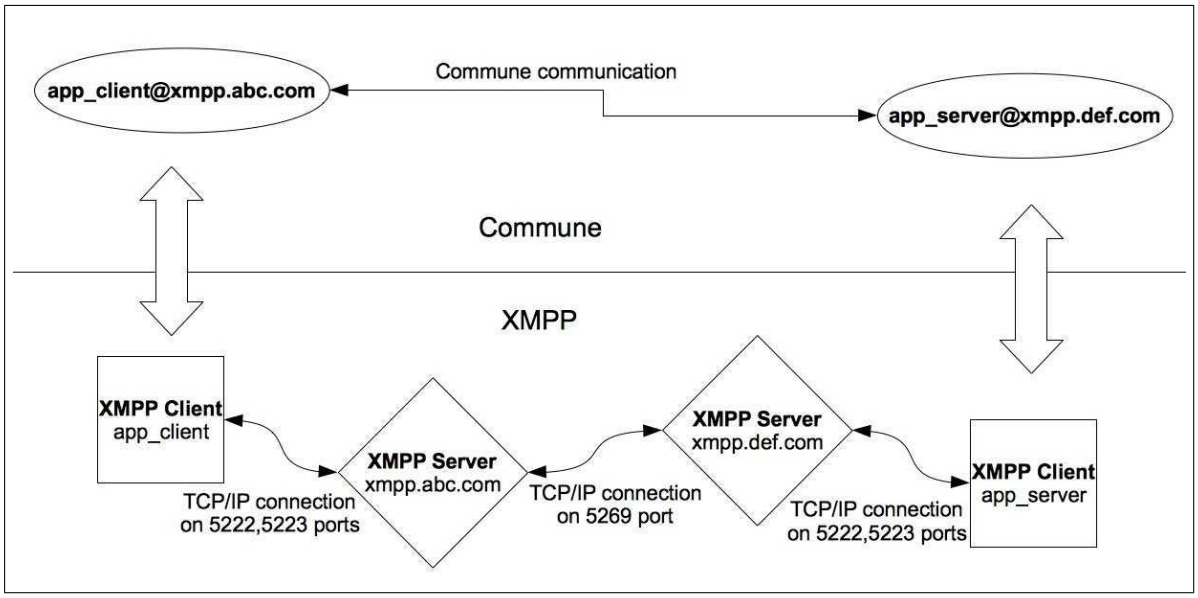

Figura 4.1: Conexão Commune *versus* Conexão XMPP

Em uma visão de alto nível, podemos representar os dois nós Commune conectados diretamente. Todavia, de fato existem várias conexões TCP/IP, pois cada cliente está conectado a um servidor e os servidores se conectam diretamente entre si. Essa arquitetura permite que

[1] http://java.sun.com/javase/downloads/index.jsp
[2] http://www.igniterealtime.org/projects/openfire

o Commune atenda ao requisito não funcional 6 - Conectividade parcial - da seção 1.3.1, pois apenas os servidores XMPP precisam ter IP público, o que facilita a configuração do *firewall*.

Para o envio e recebimento de mensagens remotas, cada nó Commune precisa estabelecer uma conexão com seu respectivo servidor XMPP. Esse passo é facilitado pelo Smack[3], uma biblioteca código aberto para clientes XMPP em Java.

Os mecanismos de segurança escolhidos para o Commune precisam manipular chaves criptográficas e certificados. Para tanto, utilizamos os pacotes *java.security* e *sun.security.provider.certpath*, que estão disponíveis por padrão na plataforma Java.

Durante o desenvolvimento do Commune foram utilizadas várias ferramentas. A IDE *Eclipse* [4] foi escolhida para a programação do código fonte e dos testes automáticos. As atividades de compilação, geração de documentação, testes e geração do arquivo *jar* foram automatizadas pela ferramenta *Maven*[5]. Os testes automáticos de unidade e de integração foram escritos com o auxílio do *framework* de testes *JUnit*[6]. Em alguns desses testes foi necessário substituir o comportamento de componentes reais por *mocks* previamente configurados. Nestes casos, utilizamos a biblioteca *EasyMock*[7].

4.2 Projeto do componente Network

Como foi dito no Capítulo 3, a solução de segurança e o protocolo de conexão do Commune foram implementados dentro do componente *Network*. Antes de mostrar os detalhes de projeto desses mecanismos, veremos como está organizada a estrutura interna do componente *Network*.

O propósito do componente *Network* é funcionar para o restante do Commune como uma interface do meio de comunicação remota. Todas as vezes que algum componente decide se comunicar com um nó remoto, ele cria uma mensagem Commune e invoca a operação *sendMessage* do componente *Network*. De modo semelhante, quando a biblioteca *Smack* recebe uma mensagem XMPP remota, o componente *Network* cria uma mensagem Commune e in-

[3] http://www.igniterealtime.org/projects/smack
[4] http://www.eclipse.org
[5] http://maven.apache.org
[6] http://www.junit.org
[7] http://easymock.org

voca o método *receiveMessage* de *Network*, que é responsável por rotear a mensagem para o componente correto no Commune.

O componente *Network* utiliza o padrão de projetos *Cadeia de responsabilidade* [14] para criar uma lista de protocolos. Cada um desses protocolos lida com um aspecto da comunicação remota entre dois nós. Durante o envio, uma mensagem Commune desce a lista de protocolos e nela são adicionados os dados pertinentes a cada protocolo. No recebimento, a mensagem sobe pela pilha sendo, assim, validada na ordem inversa do envio. Dessa forma, a implementação dos protocolos torna-se menos acoplada e a adição de novos protocolos é trivial.

Na base da pilha de protocolos do componente *Network* está a classe `XMPPProtocol`, que utiliza a biblioteca *Smack* para criar um cliente XMPP, capaz de enviar e receber mensagens pela Internet. As mensagens Commune precisam ser convertidas para mensagens XMPP. Essa conversão utiliza a serialização Java para transformar os objetos em uma cadeia de caracteres, sendo esta anexada a uma mensagem XMPP. Sobre `XMPPProtocol` estão localizadas as classes `CertificationProtocol`, `SignatureProtocol` e `ConnectionProtocol` nessa ordem.

Os mecanismos de segurança do Commune são implementados pelas classes `CertificationProtocol` e `SignatureProtocol`, que definem o protocolo de certificação e o protocolo de assinatura, respectivamente. No envio das mensagens, essas classes anexam na mensagem a chave pública e o certificado do emissor, além da assinatura da mensagem. No recebimento de uma mensagem, esses dados serão verificados e a mensagem só será aceita se eles estiverem válidos. Esses protocolos serão detalhados na seção 4.3.

A classe `ConnectionProtocol` representa a fachada do protocolo de conexão do Commune. No envio de uma mensagem, esse protocolo define os números de sessão e sequência, tanto nas mensagens de aplicação como nas mensagens de controle. No receptor, o protocolo verifica esses mesmos números, a fim de detectar se houve a perda de alguma mensagem de aplicação. Os detalhes dessa classe e das demais que formam o protocolo de conexão estão na seção 4.4.

4.3 Projeto dos mecanismos de segurança

A seção 3.2 apresenta os dois níveis de segurança definidos para integrar a solução de segurança do Commune. Cada um desses níveis é implementado por um mecanismo de segurança que possui o seu próprio protocolo no componente *Network*.

O protocolo de assinatura é responsável por assinar as mensagens enviadas e verificar a assinatura das mensagens recebidas, atestando, desta forma, a chave pública do remetente da mensagem e protegendo o sistema dos ataques de *spoofing*. Para assinar e verificar as mensagens serão necessários dois algoritmos. Primeiro, um algoritmo de *hashing* que gera um *hash* dos dados importantes da mensagem; posteriormente é aplicado o algoritmo de assinatura propriamente dito.

Durante o envio de uma mensagem, o protocolo de assinatura gera uma cadeia de caracteres com as seguintes informações:

- Endereço do emissor e do receptor da mensagem;
- Dados da invocação remota - nome do método invocado, tipos dos parâmetros do método, valores dos parâmetros;
- Números de sessão e sequência da conexão - por isto o protocolo de assinatura precisa estar localizado sob o protocolo de conexão, no componente *Network*.

Essa cadeia de caracteres é transformada em um *hash*, que é assinado pela chave privada do emissor e é anexado na mensagem. A chave pública do emissor também é anexada na mensagem.

Quando a mensagem chega no receptor, o protocolo de assinatura gera a mesma cadeia de caracteres e aplica o algoritmo de *hashing* sobre ela. O *hash* é verificado utilizando a chave pública do emissor. Se a verificação ocorrer com sucesso, então o receptor pode confiar na chave pública do emissor, pois apenas este possui a chave privada e uma assinatura só pode ser verificada através da chave pública associada à chave privada que a gerou. Se a verificação da assinatura falhar, a mensagem será descartada pelo Commune e a aplicação não será invocada.

Durante o processamento de uma mensagem, a aplicação pode consultar a API do

Commune para obter a chave pública do emissor, que poderá ser utilizada na lógica de negócio como uma identidade forte do emissor.

Na Figura 4.2 pode-se ver um diagrama de sequência que explica o processo de assinatura e verificação de uma mensagem Commune, no cenário onde a verificação ocorre com sucesso e a mensagem é entregue à aplicação no receptor.

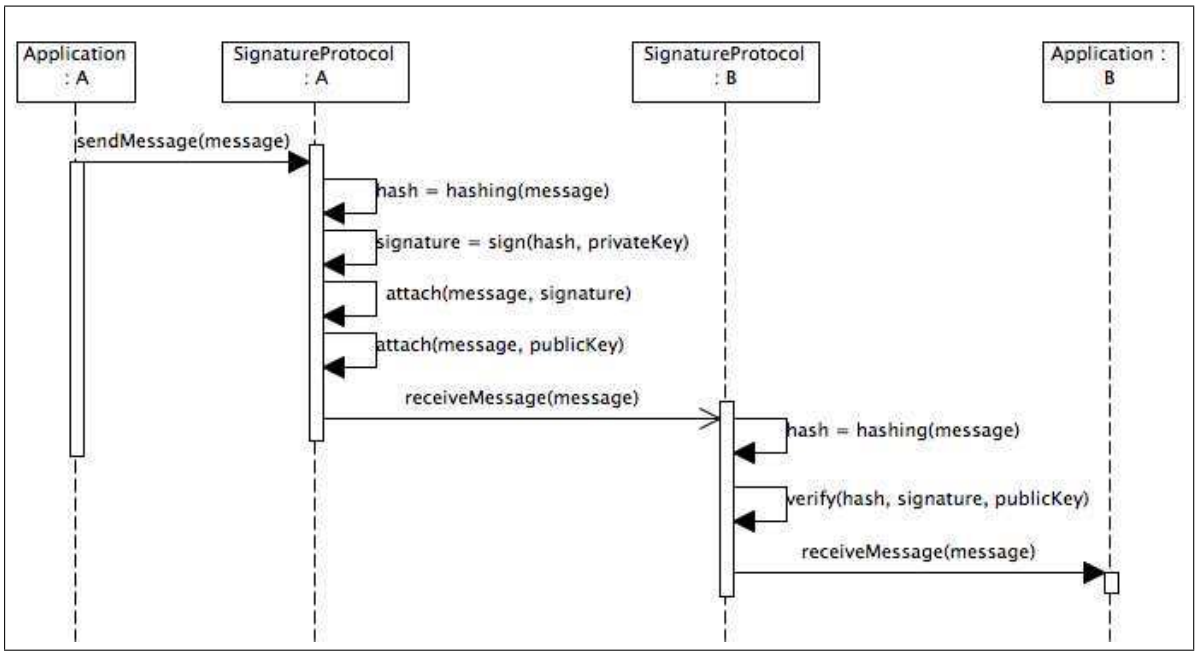

Figura 4.2: Processo de assinatura e verificação de uma mensagem Commune

Analisamos três algoritmos de *hashing*, MD5, SHA-1 e RIPEMD-160, para o protocolo de assinatura. MD5 é o algoritmo mais rápido, seguido por SHA-1 e RIPEMD-160, mas MD5 é bem menos seguro contra alguns tipos de ataque [30; 35]. Escolhemos SHA-1 pelo fato de ser rápido e seguro.

Os dois algoritmos de chave pública mais populares para a assinatura de mensagens são DSA e RSA, cujo desempenho é praticamente similar [35]. DSA é mais rápido para a assinatura das mensagens, que é feita no emissor de uma mensagem. Enquanto RSA demora menos tempo para verificar a assinatura da mensagem no receptor. Escolhemos RSA, pois além de ser utilizado para assinatura, também pode ser utilizado para, no futuro, implementar a criptografia dos dados confidenciais. O tamanho escolhido para as chaves RSA foi 1024 bits, pois é garantida, atualmente, a insuficiência de poder computacional para quebrar tal tipo de chave.

O protocolo de certificação anexa uma cadeia de certificados X.509 no envio das mensagens de abertura de conexão, que têm o número de sequência igual a zero, e verifica a consistência dessas cadeias no receptor. O certificado atesta a identidade do emissor, associando-a a sua chave pública. Sabendo que o protocolo de assinatura garante a validade da chave pública do emissor para todas as mensagens, o certificado precisa ser enviado apenas na primeira mensagem da conexão. No receptor, o protocolo de certificação mantém um mapa de chaves públicas e certificados. Quando as próximas mensagens chegarem no receptor, o mapa é consultado e o certificado é anexado na mensagem. Desse modo, a API do Commune pode ser utilizada para consultar o certificado do emissor no recebimento de qualquer mensagem.

A Figura 4.3 apresenta um diagrama de sequência que explica o mecanismo de armazenamento de certificados nos receptores de mensagens. Inicialmente é enviada uma mensagem de controle para abertura de conexão (que tem o número de sequência igual a zero). O protocolo de certificação anexa uma cadeia de certificados X.509 a essa mensagem. A cadeia de certificados atesta a identidade do emissor das mensagens.

No receptor, a consistência da cadeia de certificados é verificada pelo protocolo de certificação, apenas para a mensagem de abertura de conexão. Neste momento, o receptor utiliza um mapa para associar o certificado à chave pública do remetente. Dado que o protocolo de assinatura garante a validade da chave pública do emissor para todas as mensagens, o remetente não precisará enviar o certificado nas próximas mensagens, pois o receptor poderá recuperá-lo do mapa somente conhecendo a chave pública do remetente.

4.4 Projeto do protocolo de conexão

O modelo do protocolo de conexão do Commune foi apresentado na seção 3.3.3. A partir daquele modelo podemos definir, em termos formais, que um nó Commune representa a comunicação (conexões de saída e de entrada) com outro nó através de uma tupla $\{L, R, s, r\}$, onde:

L é o endereço do nó local,

R é o endereço do nó remoto,

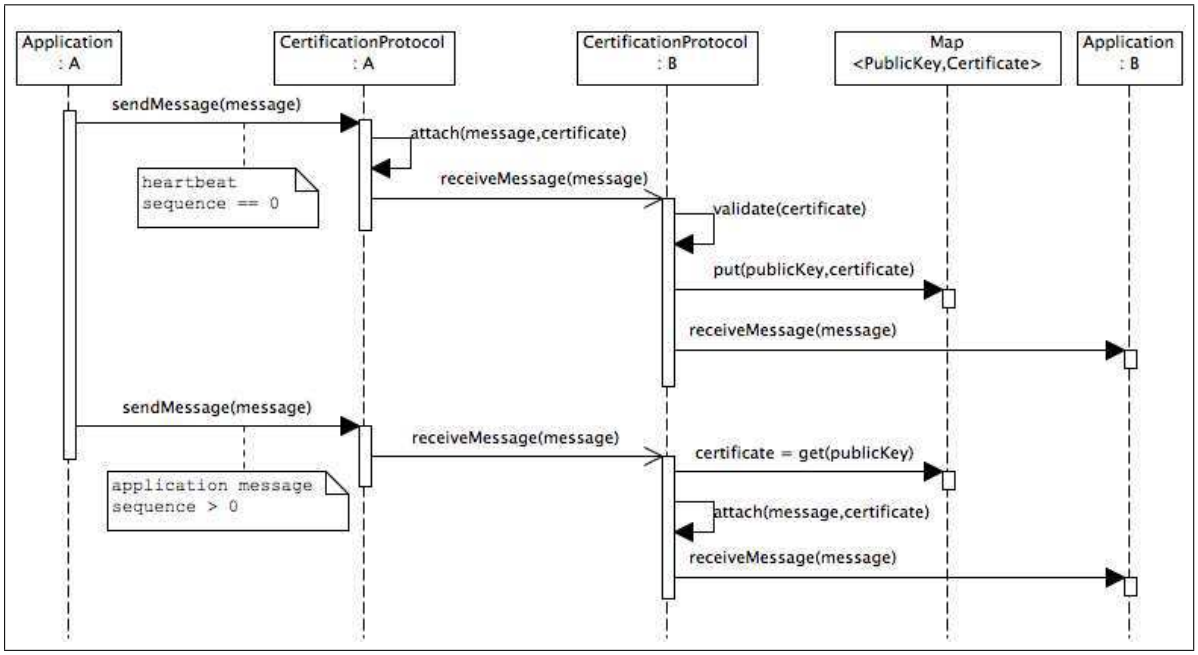

Figura 4.3: Mecanismo de armazenamento de certificados nos receptores de mensagens

s é o estado da conexão de saída, que pode assumir os valores:

Empty: o nó local não está se comunicando com o nó remoto;

Down: o nó local está interessado no nó remoto, mas o detector de falhas suspeita que o nó remoto está indisponível. Neste caso, existe um número de sessão ativo e o número sequência é zero;

Uping: o nó local está interessado no nó remoto, o detector de falhas suspeita que o nó remoto está disponível, mas a aplicação ainda não foi notificada dessa disponibilidade;

Up: o nó local está interessado no nó remoto, a aplicação já foi notificada da disponibilidade do nó remoto e está se comunicando com ele. Neste caso, existem números de sessão e sequência ativos;

Downing: o detector de falhas sinalizou a falha do nó remoto, mas a aplicação ainda não foi notificada da falha.

r é o estado da conexão de entrada, que pode assumir os valores:

Empty: o nó remoto não está se comunicando com o nó local;

R(=0): o nó remoto está interessado no nó local, mas ainda não mandou nenhuma mensagem de aplicação, então o número de sequência esperado é zero;

R(>0): o nó remoto está interessado no nó local, e já mandou alguma mensagem de aplicação, então o número de sequência esperado é maior que zero.

Quando combinamos a formalização dos estados de um nó Commune com todos os eventos que podem influenciar uma conexão (ver Tabela 3.1), geramos uma máquina de estados que auxiliou o desenvolvimento do protocolo de conexão. A Figura 4.4 mostra a evolução dos estados de um nó, quando ele inicia a conexão com o nó remoto.

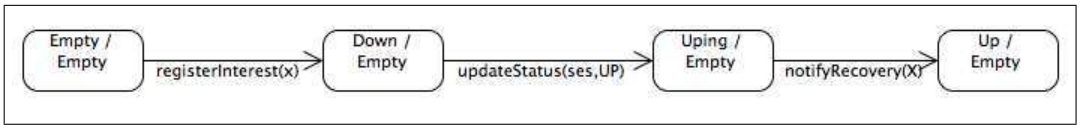

Figura 4.4: Sequência de eventos e estados para estabelecer uma conexão de saída

Partindo do estado inicial, onde nenhum dado de conexão existe, a aplicação registra o interesse no nó remoto e muda para o estado *Down/Empty*. Após receber uma mensagem de controle com a informação da disponibilidade do nó remoto (*Uping/Empty*), o detector de falhas notifica a recuperação desse nó (*Up/Empty*). Daí em diante o nó local poderá enviar mensagens para o nó remoto. Essa é uma conexão *half-duplex*, pois o nó remoto não está hábil para enviar mensagens para o nó local.

A Figura 4.5 mostra a evolução dos estados de um nó local, quando é o nó remoto que inicia a conexão.

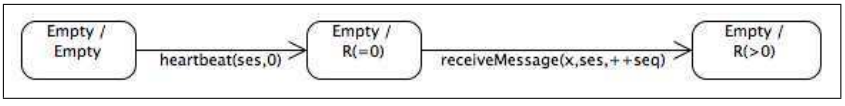

Figura 4.5: Sequência de eventos e estados para estabelecer uma conexão de entrada

Partindo do estado inicial, o detector de falhas local recebe uma mensagem de controle indicando o início de uma conexão e o estado é transicionado para *Empty/R(=0)*. Quando o nó local receber mensagens de aplicação do nó remoto, o número de sequência da conexão de entrada será maior que zero (*Empty/R(>0)*). Novamente, essa é uma conexão *half-duplex*, pois o nó local não está hábil para enviar mensagens para o nó remoto.

Ao combinamos as duas máquinas de estado descritas acima, obtemos uma nova máquina de estados que contêm todos os caminhos possíveis e todos os estados intermediários até se atingir o estado que representa uma conexão *full-duplex* (*Up/R(>0)*). Esse é o estado mais comum para a comunicação entre dois nós, pois nele as mensagens de aplicação estarão fluindo nos dois sentidos da comunicação. A Figura 4.6 mostra a máquina de estados combinada.

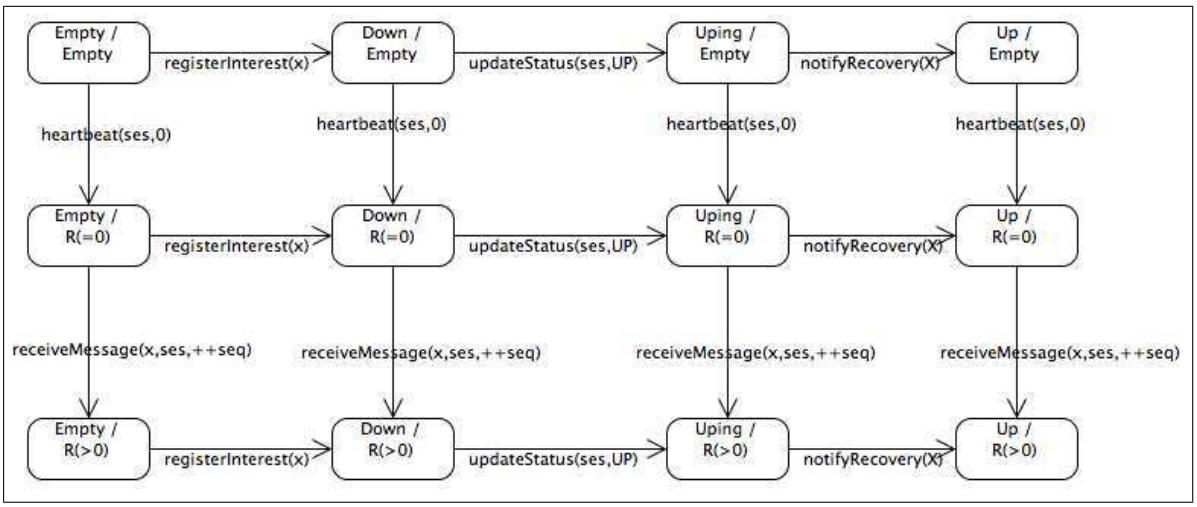

Figura 4.6: Combinação dos estados identificados

Além das sequências de eventos para o estabelecimento das conexões de saída e de entrada, modelamos os eventos de ocorrência de falhas. Essas falhas alteram o estado da comunicação entre dois nós, para que a aplicação possa tratá-las e a conexão seja restabelecida ou descartada. As falhas detectadas pelo protocolo de conexão do Commune e o significado de cada uma delas são especificados a seguir:

- O nó receptor espera um determinado número de sessão, mas recebe um *heartbeat* com sessão diferente - indica que o emissor falhou ou reiniciou a comunicação;
- O nó receptor espera um número de sequência igual a zero, mas recebe um *heartbeat* com sequência diferente de zero - houve perda de mensagem de aplicação;
- O nó receptor espera um determinado número de sessão, mas recebe uma mensagem de aplicação com sessão diferente - indica que o emissor falhou ou reiniciou a comunicação;

- O nó receptor espera um determinado número de sequência, mas recebe uma mensagem de aplicação com sequência diferente da esperada incrementada por um - houve perda de mensagem de aplicação;
- Ocorreu um estouro no tempo de detecção de falhas - o nó emissor suspeita a falha do nó receptor;
- O nó emissor recebe um *updateStatus* indicando que o nó remoto está indisponível;
- O nó emissor espera um determinado número de sessão, mas recebe um *updateStatus* com sessão diferente - indica que o emissor falhou ou reiniciou a comunicação.

A Figura 4.7 mostra uma máquina de estados com as transições de estado quando uma falha é detectada. Para facilitar a visualização das transições, agrupamos todos os tipos de falhas em um evento chamado *failure detected*, cujo destino sempre será *Empty/Empty*, *Down/Empty* ou *Downing/Empty*. Além disso, nessa máquina de estados podem ser vistas as transições que representam o evento de desconexão (*release*), o qual sempre modificará o estado da comunicação para *Empty/Empty*.

Essa máquina de estados mostra um novo estado, chamado *Downing/Empty*, utilizado para representar o momento em que o nó emissor suspeita que o receptor está disponível (*Uping* ou *Up*) e a sua falha acaba de ser notada pelo Commune, todavia a aplicação ainda não foi notificada. Existem duas saídas para esse estado, quando a aplicação é notificada da falha ou quando coincidentemente a aplicação se desconecta do nó remoto.

Após a modelagem de todos os eventos da máquina de estados de um nó Commune, conseguimos identificar a existência de 13 estados com comportamentos diferentes diante dos eventos de uma conexão. Para tornar o código do protocolo de conexão mais legível, decidimos utilizar o padrão de projeto *Estado* [14], onde cada estado é implementado por uma classe e os eventos são representados por métodos de uma interface que deve ser implementada por todos os estados. A Figura 4.8 apresenta um diagrama de classes com as classes e interfaces que compõem o protocolo de conexão.

A principal ligação do protocolo de conexão com o Commune é a classe `ConnectionProtocol`, que implementa os métodos abstratos da classe `Protocol`. Essa classe é a superclasse de todos os protocolos do componente

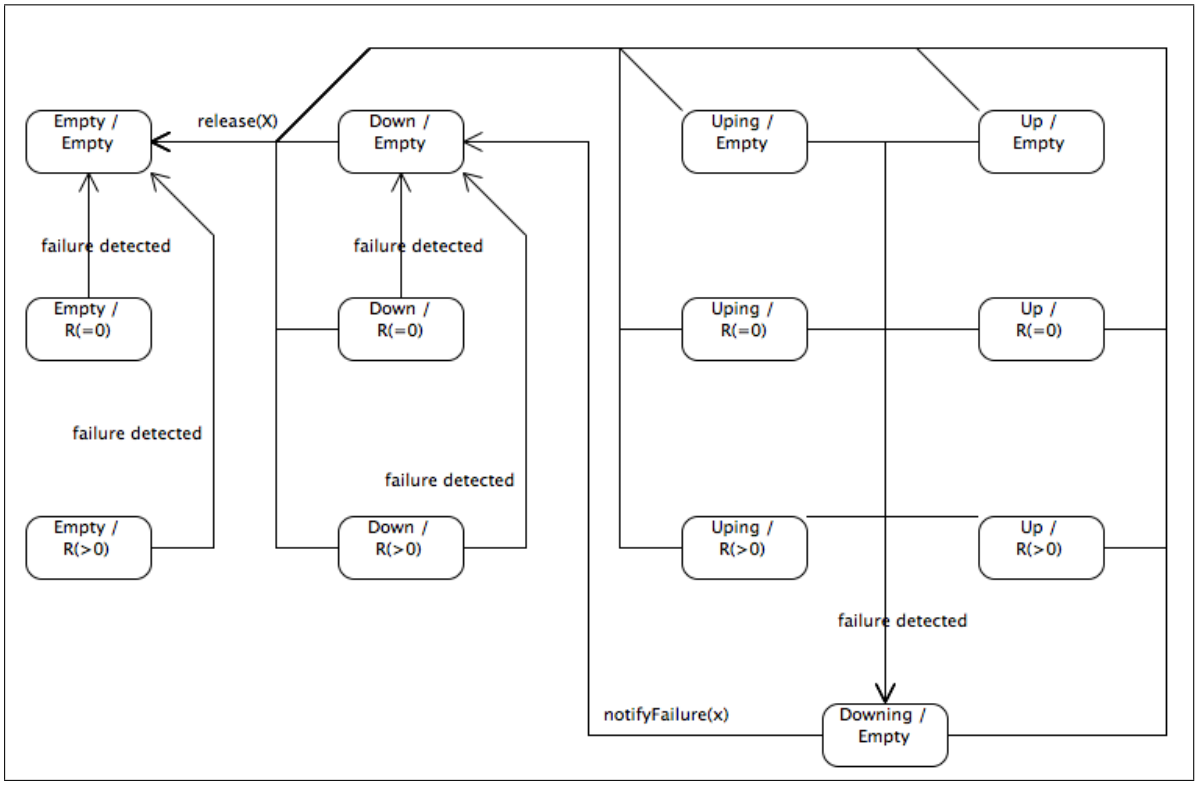

Figura 4.7: Transições de falhas e desconexão

Network. Desta forma, no envio de cada mensagem será invocado o método `ConnectionProtocol.onSend(Message)` e na recepção uma mensagem o Commune invoca o método `ConnectionProtocol.onReceive(Message)`. A classe `ConnectionProtocol` possui apenas o código necessário para ligar o protocolo de conexão ao componente *Network*. De fato, a maior parte da lógica de conexão se encontra na classe `ConnectionManager`, que além de tratar o envio e recebimento de mensagens, também atua como *Listener* de outros eventos de conexão que podem acontecer dentro do Commune:

- A aplicação registra interesse por um nó remoto – evento capturado através do método `StubListener.stubCreated(StubReference);`
- A aplicação desregistra o interesse por um nó remoto – evento capturado através do método `StubListener.stubReleased(ServiceID);`
- Estouro do tempo limite de conexão (*detection time*) com um nó remoto – evento capturado pelo método `TimeoutListener.timeout(ServiceID);`

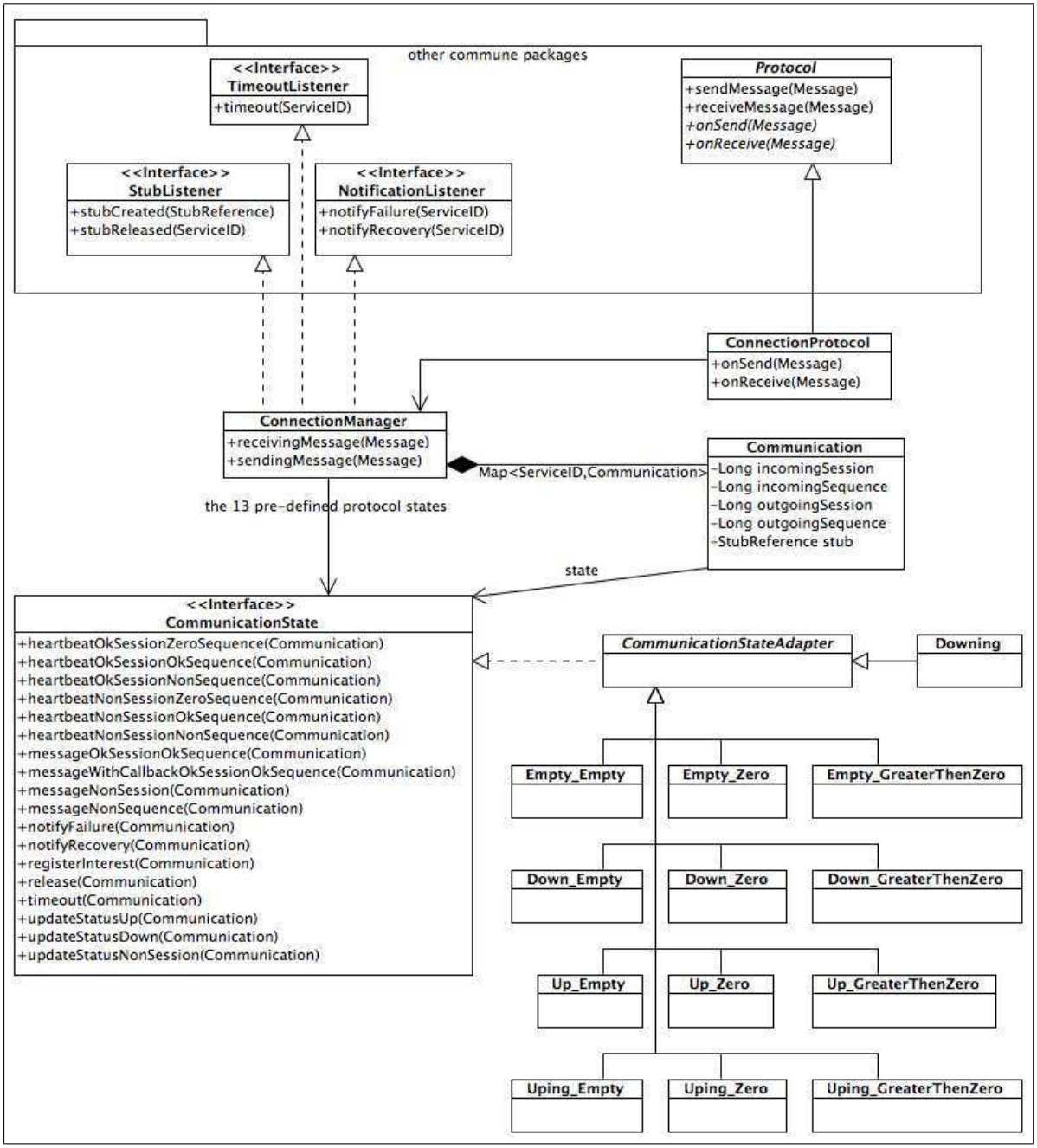

Figura 4.8: Diagrama de classes do protocolo de conexão do Commune

- A aplicação é notificada da falha de um nó remoto – evento capturado através do método `NotificationListener.notifyFailure(ServiceID);`
- A aplicação é notificada da recuperação de um nó remoto – evento capturado através do método `NotificationListener.notifyRecovery(ServiceID);`

Alguns dos métodos no diagrama de classes utilizam como parâmetro as classes `ServiceID` e `StubReference`. `ServiceID` representa o endereço de um serviço remoto e se baseia na estrutura de endereçamento do XMPP. `StubReference` contém todos os dados de um *stub* de um nó remoto, o endereço (`ServiceID`), o estado de disponibilidade do nó remoto e a referência para o *stub* propriamente dito.

A classe `ConnectionManager` cria um objeto do tipo `Communication` toda vez que se cria uma conexão com um objeto remoto, e o armazena em um mapa cuja chave é o `ServiceID` do objeto remoto. O objeto `Communication` contém: os dados das conexões de saída (*outgoing*) e de entrada (*incoming*); a referência para o *stub*; e o estado atual da comunicação.

A lógica de transição de estados de uma conexão foi implementada nas 13 classes de estado, conforme o padrão de projeto *Estado*. Essas classes são *singletons* criadas pela `ConnectionManager`. Todos os métodos das classes de estado recebem um objeto `Communication` como parâmetro, atuam sobre esse objeto, podendo até alterar o estado da comunicação. Após a execução do método, o objeto de estado não tem mais ciência dos parâmetros, podendo, então, ser reutilizado no processamento de transições de outros objetos `Communication`.

A funcionalidade da classe `ConnectionManager` pode ser entendida como o mapeamento dos 7 eventos representados pelos seus métodos públicos para os 18 da interface `CommunicationState`. O mapeamento pode ser visto na listagem abaixo:

- O recebimento de uma mensagem remota (`receivingMessage(Message)`) pode demandar a criação de um novo objeto `Communication` e pode ser mapeado em um dos eventos:
  - `heartbeatOkSessionZeroSequence` - sessão igual à esperada, sequência igual a zero;
  - `heartbeatOkSessionOkSequence` - sessão igual à esperada, sequência igual à esperada;
  - `heartbeatOkSessionNonSequence` - sessão igual à esperada, sequência diferente da esperada;

  - `heartbeatNonSessionZeroSequence` - sessão diferente da esperada, sequência igual a zero;
  - `heartbeatNonSessionOkSequence` - sessão diferente da esperada, sequência igual à esperada;
  - `heartbeatNonSessionNonSequence` - sessão diferente da esperada, sequência diferente da esperada;
  - `messageWithCallbackOkSessionOkSequence` - mensagem com *callback*, sessão igual à esperada, sequência igual à esperada;
  - `messageOkSessionOkSequence` - mensagem sem *callback*, sessão igual à esperada, sequência igual à esperada;
  - `messageNonSession` - mensagem de qualquer tipo, sessão diferente da esperada;
  - `messageNonSequence` - mensagem de qualquer tipo, sequência diferente da esperada;
  - `updateStatusUp` - o nó remoto está disponível;
  - `updateStatusDown` - o nó remoto está indisponível;
  - `updateStatusNonSession` - sessão diferente da esperada.

- A criação de um *stub* (`stubCreated(StubReference)`) pode demandar a criação de um novo objeto `Communication` e será mapeada no evento `registerInterest` - o nó local deseja abrir conexão com o nó remoto;
- A remoção de um *stub* (`stubReleased(ServiceID)`) será mapeada no evento `release` - o nó local deseja encerrar a conexão com o nó remoto - e removerá o objeto `Communication` do mapa de comunicações remotas;
- O estouro do tempo de conexão (`timeout(ServiceID)`) será mapeado no evento `timeout` - suspeita de falha por parada do nó remoto, após passar um período maior do que o *detection time* sem responder mensagens de controle;
- Quando a aplicação recebe a notificação da suspeita de falha de um nó remoto (`notifyFailure(ServiceID)`), será ativado o evento `notifyFailure`;

- Quando a aplicação recebe a notificação da recuperação de um nó remoto (`notifyRecovery(ServiceID)`), será ativado o evento `notifyRecovery`;

- O envio de uma mensagem remota (`sendingMessage(Message)`) não será mapeado para os eventos de estado, pois é utilizado apenas para definir os números de sessão e sequência das mensagens.

Na criação de um objeto `Communication`, o estado inicial será *Empty_Empty* e o objeto será armazenado no mapa de comunicações remotas.

As Seções 5.1 e 5.2 mostram a avaliação desse modelo, em relação ao atendimento dos predicados da semântica de detecção de falhas definida neste trabalho.

4.4.1 Protocolo de conexão do Commune *versus* TCP

Encerraremos esta seção fazendo uma breve comparação do protocolo de conexão do Commune com o protocolo TCP [31]. Essa comparação é necessária, pois uma das soluções para implementar a semântica de detecção de falhas do Commune seria reimplementar o modelo TCP, que é capaz de detectar perda de mensagens e já foi amadurecido por quase três décadas de uso em escala global.

TCP é um protocolo que se localiza entre a aplicação e o protocolo de mais baixo nível IP. Diversos comportamentos imprevisíveis da rede sobre IP, como por exemplo, congestionamento e alterações de roteamento para balanceamento de carga, podem fazer com que as mensagens IP sejam perdidas, duplicadas, ou entregues fora de ordem. O protocolo TCP é capaz de detectar esses problemas, reagindo com retransmissão de pacotes, rearranjo de pacotes fora de ordem, e controle de congestionamento. Pelo fato de possuir todos esses mecanismos, TCP é um protocolo bastante complexo e com muitos estados possíveis, como pode ser visto no trabalho de Zaghal e Khan [39].

Analogamente, o Commune se localiza entre a aplicação e o protocolo XMPP. Como já foi visto nesta dissertação, a semântica de falhas do Commune precisa apenas sinalizar um erro quando detecta a perda de uma mensagem. Portanto, não será necessária a retransmissão das mensagens. Além disto, o Commune não precisa tratar o congestionamento de mensagens, pois em projetos anteriores a equipe do OurGrid alterou a biblioteca *Smack* para implementar essa funcionalidade, e a especificação de XMPP [33] obriga os servidores

XMPP a garantirem a entrega de mensagens em ordem. Diante deste cenário, decidimos implementar um novo modelo mais simples para o protocolo de conexão do Commune.

4.5 Exemplos de uso

Esta seção mostra alguns exemplos de uso do Commune, tais quais: a criação de um *container* Commune; a publicação de um objeto local; o registro de interesse em objetos remotos; as notificações de recuperação e falha de objetos remotos; e a invocação de métodos remotos.

Criação de um *container* Commune

No Commune um *container* também é chamado de módulo e é representado pela classe *Module*. Antes de criar um módulo Commune, o usuário precisa inicializar um objeto de contexto (*ModuleContext*), que possui um mapa com as propriedades demandadas pelo Commune, como nome de usuário, senha e servidor XMPP, par de chaves criptográficas, entre outros.

No trecho de código abaixo, o contexto é inicializado a partir dos dados de um arquivo de propriedades chamado *application.properties*. Posteriormente, o contexto é utilizado para criar o módulo, que recebe o nome de *MODULE_NAME*.

```
DefaultContextFactory contextFactory =
    new DefaultContextFactory(
        new PropertiesFileParser("application.properties"));
ModuleContext context = contextFactory.createContext();
Module module = new Module("MODULE_NAME", context);
```

Publicação de um objeto local

Após a criação de um módulo Commune, o programador pode publicar objetos locais, a fim de que esses possam ser invocados por outros módulos. Um objeto local precisa herdar de uma interface que atenda aos seguintes requisitos:

- Seja anotada com *@Remote*;
- Não possua métodos com tipo de retorno diferente de *void*;
- Não possua métodos que lancem exceção.

Essa interface é denominada Interface Remota e é utiliza nos módulos cliente para gerar os *stubs*. O código a seguir mostra um exemplo de interface remota e como ela pode ser implementada em um objeto local.

```
@Remote
public interface MyInterface {
    void ping();
}

public class MyObject implements MyInterface {
    public void ping() {
        System.out.println("Receiving a ping");
    }
}
```

Cada objeto local precisa ter um nome único dentro de um módulo Commune. Assim sendo, quando um objeto local é publicado, ver código abaixo, um nome precisa ser fornecido.

```
module.getContainer().deploy("SERVICE_NAME", new MyObject());
```

Registro de interesse em objetos remotos

Os trechos de código que foram mostrados anteriormente se localizam no lado servidor de uma comunicação Commune. No lado cliente, é preciso registrar o interesse no objeto remoto, a fim de que se obtenha uma referência para o mesmo, que possa ser invocada pela aplicação. Para o registro de interesse, o cliente deve utilizar o objeto do tipo *ServiceManager*, que é provido automaticamente pelo Commune, e precisará fornecer as seguintes informações:

- Nome do monitor - um objeto local que será notificado pelo detector de falhas quando ocorrer a recuperação e a falha do objeto remoto;
- O endereço do objeto remoto, no formato:
  - usuário_XMPP@servidor_XMPP/nome_módulo/nome_objeto;

- Tipo da interface remota que o objeto remoto implementa.

O código a seguir exemplifica o registro de interesse.

```
manager.registerInterest("MONITOR_NAME",
    "app@xmpp.ourgrid.org/MODULE_NAME/SERVICE_NAME",
    MyInterface.class);
```

Notificações de recuperação e falha de objetos remotos

Após o código de registro de interesse, o programador deve implementar o objeto monitor, responsável pelas ações que devem ser feitas quando o objeto remoto se tornar disponível (recuperação) ou indisponível (falha). Obrigatoriamente, o monitor deve possuir dois métodos de notificação, um para a recuperação e outro para falha. Esses métodos devem (i) ter tipo de retorno *void*, (ii) possuir apenas um parâmetro com o mesmo tipo da interface remota do objeto remoto e (iii) não lançar exceção.

O método de notificação de recuperação deve ser anotado com *@RecoveryNotification* e o método de notificação de falha com *@FailureNotification*, como pode ser visto neste código:

```
public class MyMonitor {
    @RecoveryNotification
    public void serverIsUp(MyInterface server) {
        System.out.println("Server is up");
        (...)
    }
    @FailureNotification
    public void serverIsDown(MyInterface server) {
        System.out.println("Server is down");
    }
}
```

Uma das ações que o monitor pode fazer após uma notificação é encerrar a conexão com o objeto remoto. Se o monitor não fizer isto, o cliente permanecerá interessado no objeto

remoto indefinidamente e haverá sempre a alternância das notificações de recuperação e de falha.

Invocação de métodos remotos

Após a notificação de recuperação e antes da notificação de falha, um objeto remoto é marcado como disponível e pode ser utilizado para o cliente efetuar uma invocação de método remoto. De fato, o parâmetro que é recebido pelo método de notificação de recuperação é um *stub* do objeto remoto e transforma as invocações de método locais em mensagens que são enviadas pela Internet até o servidor.

```
@RecoveryNotification
public void serverIsUp(MyInterface server) {
    System.out.println("Server is up");
    server.ping();
}
```

Se o *stub* for invocado quando o objeto remoto está indisponível, o Commune lançará uma exceção para a aplicação cliente.

O próximo Capítulo trata da validação formal, avaliação de desempenho e dos testes no Commune.

Capítulo 5

Validação e Testes

Este Capítulo contém as avaliações e testes realizados para o sistema de *middleware* Commune. Inicialmente, provaremos que o protocolo de conexão do Commune atende aos predicados de uma semântica de detecção de falhas abrangente, definidos na seção 3.3.2. Como um dos predicados precisou de uma ferramenta formal para ser provado, detalharemos como foi o processo de validação formal. Adiante, descreveremos como foi feita a avaliação de desempenho do Commune. Finalizamos o Capítulo, relatando o teste de uso do Commune em um ambiente de produção.

5.1 Prova do atendimento dos predicados de detecção de falhas

Na seção 3.3.2 definimos os predicados de uma semântica de detecção de falhas abrangente, capaz de detectar as falhas por parada dos nós e falhas por omissão decorrentes da perda de mensagens. Agora provaremos como o protocolo de conexão do Commune, cujo projeto foi detalhado na seção 4.4, atende a todos esses predicados.

Para nos auxiliar nessa prova, usaremos um artifício semelhante ao que Chandra e Toueg utilizaram no seu trabalho com detectores de falhas imperfeitos [6]. Eles definiram propriedades que deveriam ser verdadeiras para sempre. Todavia, esse tipo de afirmativa é improvável para sistemas reais, que podem ser afetados por quedas de energia, desconectividade da rede de comunicação, quebras de hardware, entre outros. Além disto, no caso dos sistemas

assíncronos, é impossível determinar se um processo remoto realmente parou ou se está temporariamente sobrecarregado, portanto é normal que haja falsas suspeitas de falhas. Chandra e Toueg mostram que, de fato, é necessário que as propriedades sejam verdadeiras apenas por um período de tempo longo o suficiente para que as aplicações reais possam avançar na sua computação. Exemplificando, podemos assumir que:

- Um nó A ficará disponível por um período de tempo suficientemente longo;
- Não haverá perda de mensagens no canal de comunicação por um período de tempo suficientemente longo;
- Um nó remoto não ficará sobrecarregado por um período de tempo suficientemente longo, portanto, neste período, responderá às mensagens de controle dentro do *timeout* e, consequentemente, o detector de falhas não levantará falsas suspeitas por um período de tempo suficientemente longo.

Assim sendo, nas provas a seguir, assumiremos que $G(p)$ significa que no futuro a proposição p será verdade por um período de tempo suficientemente longo, para que a aplicação possa avançar na sua computação.

Além disso, nas provas também faremos referência aos estados que um nó Commune pode assumir (ver Figuras 4.6 e 4.7).

O Predicado 1 trata do estabelecimento e da manutenção de uma conexão Commune, e foi formalizado pelas seguintes expressões:

(i) Se *G(A e B e A » B e !(A $\not$> B))*, então *F(A −> B)*

(ii) Se *(A −> B e G(A) e G(B) e !F(A $\not$> B))*, então *G(A −> B)*

A expressão (i) indica que, por um período suficientemente longo, A e B estarão disponíveis, A estará interessado em B e que não ocorrerá perda de mensagens. Portanto, A poderá estar nos estados *(Downing), (Down / *), (Uping / *), (Up /*)*, nos quais permanecerá enviando mensagens de `heartbeat` para B. Desconsideraremos os estados *(Downing) e (Uping / *)* pois, após a notificação de falha ou recuperação, o estado do componente será migrado automaticamente para *(Down / *) e (Up / *)* respectivamente. No lado oposto da comunicação, B pode estar em qualquer estado e, mesmo que ocorra alguma falha por causa dos `heartbeats` de A, em algum momento B passará a responder com

`updateStatus(UP)` dentro do período de *detection time* e, daí em diante, não haverá *timeout*.

Se atualmente B aparece como indisponível para A *(Down / *)*, então, na próxima resposta de controle, A será notificado da disponibilidade de B e seu estado será mudado para *(Up / *)*. Se atualmente B está disponível para A *(Up / *)*, então B continuará respondendo com `updateStatus(UP)` e permanecerá disponível. Nesses dois casos, a expressão *F(A −> B)*, que significa – em algum momento no futuro A estará conectado a B *(Up / *)* – é verdade.

Na expressão (ii), A já está conectado a B. A e B estarão disponíveis e não haverá perda de mensagens de aplicação no meio de comunicação, por um período suficientemente longo. Neste cenário, A enviará mensagens de `heartbeat` periodicamente para B. Como não há perda de mensagens de aplicação no canal, os números de sequência das mensagens de controle e de aplicação serão iguais aos esperados. Dado que B está disponível e não há perda de mensagens no canal, as mensagens de `updateStatus(UP)` chegarão em A dentro do período de *detection time* e não haverá *timeout*. Elimina-se, portanto, as duas maneiras de haver detecção de falhas neste cenário e a conexão de A para B permanecerá ativa (*G(A −> B)*), por um período suficientemente longo.

A detecção de falha por parada dos nós é expressa no Predicado 2:

Se *(A −> B e F(b))*, então *F(A +> b)*

A está conectado a B *(Up / *)* e B se torna indisponível. Logo, A deixará de receber as mensagens de `updateStatus(UP)` oriundas de B. Após um período de tempo próximo de *detection time*, ocorrerá um *timeout* e A será notificado da suspeita da falha de B *(Downing)* e *(Down / Empty)*.

O Predicado 3 – detecção de falha por omissão devido a perda de mensagem – é representado pelas duas expressões a seguir:

(i) Se *(A −> B e F(A +> B))*, então *F(A +> b)*

(ii) Se *(A −> B e B −> A e (F(A +> B) ou F(B +> A)))*, então *F(A +> b) e F(B +> a)*

Na expressão (i), A está conectado a B *(Up / *)* e ocorre uma perda de mensagem de aplicação. Quando a próxima mensagem for enviada de A para B, o número de sequência estará diferente do esperado. Portanto, B invalidará a conexão de entrada *(* / Empty)* e não responderá mais com mensagens de `updateStatus(UP)`. Após um período de tempo

próximo de *detection time*, ocorrerá um *timeout* e A será notificado da suspeita da falha de B *(Downing)* e *(Down / Empty)*.

A expressão (ii) trata do cenário de conexões reversas, onde A está conectado a B *(Up / *)* e B está conectado a A *(Up / *)*. Daí, ocorre uma perda de mensagem de aplicação em qualquer um dos sentidos. No entanto, como essa expressão possui propriedades simétricas em relação ao sentido da conexão, assumiremos que a mensagem foi perdida no sentido de A para B. Portanto, quando a próxima mensagem for enviada de A para B, o número de sequência estará diferente do esperado e B invalidará as conexões de saída e de entrada *(Downing)*. A aplicação em B será notificada da falha de A *(Down / Empty)* e B não responderá mais com mensagens de `updateStatus(UP)` para A. Após um período de tempo próximo de *detection time*, ocorrerá um *timeout* e A será notificado da suspeita da falha de B *(Downing)* e *(Down / Empty)*.

O Predicado 4 é mais genérico do que os outros, pois trata do restabelecimento de uma conexão. Esse processo pode atravessar quaisquer estados na comunicação entre dois nós. Portanto, a prova manual desse predicado poderia ser demasiadamente extensa. A seção 5.2 detalha o processo de validação formal utilizado para provar o atendimento desse predicado.

5.2 Validação formal

O Predicado 4 da semântica de detecção de falhas do Commune trata do restabelecimento de uma conexão e é representado pela expressão:

Se *(P(A −> B) e A +/> b e G(A e B e A » B e !(A +/> B)))*, então *F(A −> B)*

Dado que A está conectado a B e, posteriormente, acontece alguma falha nessa conexão de modo que A visualiza B como indisponível, se, por um período suficientemente longo, A e B se mantiverem disponíveis e A permanecer interessado por B, então em algum momento do futuro será possível restabelecer a conexão de A para B.

Na sequência de eventos (A −> B), (A +/> b) e (A −> B), o cenário avaliado pode passar pela maior parte dos estados possíveis da comunicação entre dois nós Commune. Se houver algum defeito, o protocolo de conexão pode entrar em *deadlock* e a conexão não conseguirá ser restabelecida. Com a finalidade de provar que esse predicado pode ser atendido, analisamos todos os estados possíveis na comunicação entre dois nós Commune.

O estado da comunicação entre dois nós, A e B, é composto pelo estado de A, estado de B, e pelas mensagens que estão trafegando nos quatro canais de controle e nos dois canais de dados que compõem as conexões reversas (ver Figura 3.4). O propósito do Predicado 4 é mostrar que não existe um estado neste cenário que cause um *deadlock* na comunicação, impedindo assim o restabelecimento da conexão entre os dois nós.

O cenário da comunicação entre dois nós Commune pode gerar um grande número de estados, inviabilizando a validação manual do Predicado 4. Assim sendo, decidimos utilizar técnicas de verificação de modelos (do inglês *model checking*) para auxiliar na obtenção dessa prova. Segundo Clarke e Emerson [8], a verificação de modelos é uma técnica automática que, dado um modelo de estados finito de um sistema e uma propriedade lógica, verifica sistematicamente se a propriedade se aplica ao modelo, a partir de um estado inicial.

Um modelo finito para o cenário em estudo pode ser obtido ao limitar o tamanho dos canais de comunicação. E a propriedade a ser avaliada pode ser extraída a partir da expressão temporal que representa o Predicado 4. O espaço de estados gerado cresce exponencialmente em relação ao tamanho do modelo validado, portanto unificamos os dois canais de saída de uma conexão Commune (dados e controle), a fim de reduzir o tamanho do espaço de estados. Deste modo, o modelo avaliado foi composto por:

- Dois nós Commune - cada um com uma máquina de estados que pode assumir qualquer um dos 13 estados possíveis (ver Figuras 4.6 e 4.7);
- Dois canais principais - originados da junção dos canais de saída (dados e controle), os quais podem estar vazios ou preenchidos com as seguintes mensagens (ver Tabela 3.1):
    1. *heartbeat(ses,zero)*
    2. *heartbeat(ses,seq)*
    3. *heartbeat(ses,!seq)*
    4. *heartbeat(!ses,zero)*
    5. *heartbeat(!ses,seq)*
    6. *heartbeat(!ses,!seq)*
    7. *receiveMessage(ses,++seq)*

8. *receiveMessage(ses,!++seq)*
9. *receiveMessage(!ses,++seq)*
10. *receiveMessageWithCallback(ses,++seq)*
11. *receiveMessageWithCallback(ses,!++seq)*
12. *receiveMessageWithCallback(!ses,++seq)*

- Dois canais reversos - originados do canal de controle de entrada, os quais podem estar vazios ou preenchidos com as seguintes mensagens:

1. *updateStatus(ses,UP)*
2. *updateStatus(ses,DOWN)*
3. *updateStatus(!ses,*)*

A Figura 5.1 representa visualmente os elementos desse modelo:

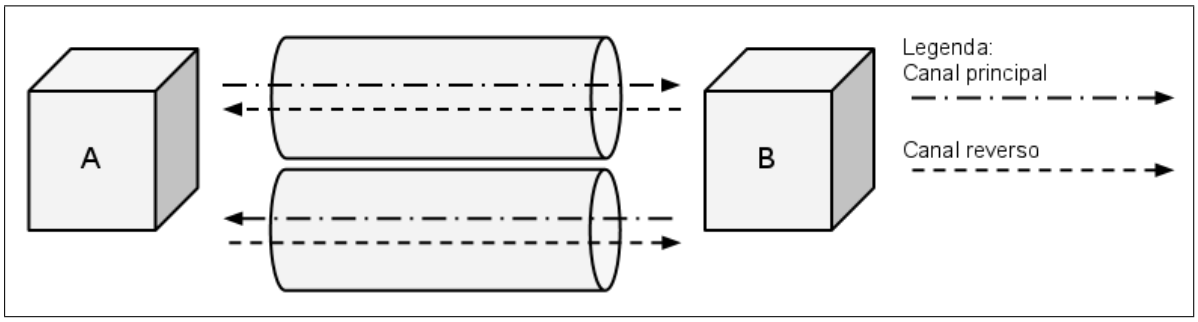

Figura 5.1: Modelo finito da comunicação entre dois nós Commune

Combinando-se os 13 estados possíveis dos nós, com os 13 estados nos canais principais (12 tipos de mensagem mais o canal vazio) e os 4 estados dos canais reversos (3 tipos de mensagem mais o canal vazio), concluímos que o tamanho do espaço de estados é:

$13^2 * 13^2 * 4^2 = 456.976$

Para testar o Protocolo de Conexão do *Commune*, escolhemos a ferramenta *Spin* [18], devido a dois motivos principais. Em primeiro lugar, para definir os seus modelos, o *Spin* utiliza a linguagem *Promela* [19], que se adaptou bem ao modelo de nós com máquinas de estado e canais de comunicação utilizado neste trabalho. Em segundo lugar, o *Spin* é uma ferramenta amadurecida durante vinte anos de uso e possui diversas técnicas para otimizar o desempenho das suas verificações.

Assim sendo, o cenário em estudo foi modelado na linguagem Promela, exatamente como aparece na Figura 5.1. O código fonte desse modelo pode ser visto no Apêndice A. No entanto, a linguagem *Promela* não possui semântica para verificar o tipo de assertiva do predicado 4 numa única execução. Portanto, foi necessário utilizar um *script shell* para gerar os modelos dos 456.976 estados possíveis e para verificá-los individualmente. Em cada uma das verificações, não houve *deadlocks* e todos os demais estados foram atingidos.

Concluímos que, independente do estado da comunicação entre dois nós Commune, sempre será possível restabelecer a conexão entre os nós. Logo, não existem *deadlocks* no protocolo de conexão e o predicado 4 é atendido pelo modelo proposto.

5.3 Avaliação de desempenho

Após sua implementação, o Commune foi submetido a uma avaliação de viabilidade com o intuito de estudar se as soluções introduzidas nesse sistema de *middleware*, principalmente a solução de segurança e o protocolo de conexão, gerariam uma sobrecarga proibitiva, inviabilizando o uso do Commune em ambientes de produção. Essa avaliação consistiu de um experimento ponto-a-ponto entre dois computadores geograficamente distribuídos, comunicando-se via Commune e outra tecnologia de referência.

Um computador estava localizado na rede interna da Universidade Federal de Campina Grande/BR, conectada ao *backbone* da RNP, e o outro foi alugado da empresa Linode, localizada em New Jersey/EUA. Foram instalados um nó emissor (BR) e outro receptor (EUA) para a comunicação remota entre os computadores. O experimento consistia do envio de mensagens do emissor para o receptor, que as respondia imediatamente. O experimento comparou o tempo decorrido desde o envio de cada mensagem inicial até o recebimento de cada resposta no emissor.

Como referência, escolhemos a tecnologia Java RMI, pela sua maturidade e popularidade no contexto dos sistemas distribuídos. Portanto, criamos os nós emissor e receptor em RMI, conectados diretamente um ao outro. Da mesma forma, instalamos os nós emissor e receptor em Commune. Esses não se conectavam diretamente e cada um utilizava um servidor XMPP diferente, sendo que o servidor do emissor estava em um terceiro computador na mesma rede local, enquanto o servidor do receptor estava no mesmo computador deste.

Para cada uma das tecnologias testadas (em experimentos separados) foram enviadas 500 mensagens pelo emissor. O tempo médio, a mediana e o desvio padrão para RMI foram $255ms$, $254ms$ e $16,2ms$, respectivamente. No Commune, o tempo médio foi $1256ms$, a mediana $1136ms$ e o desvio padrão $359ms$. Observamos que o Commune possui uma latência cerca de cinco vezes maior que a latência apresentada por RMI, uma tecnologia de referência para o desenvolvimento de sistemas distribuídos sobre a Internet.

Um dos propósitos do Commune é permitir a manutenção de diversas conexões simultaneamente. Sendo que essa característica é mais importante do que a latência das mensagens entre os componentes remotos. Deste modo, embora a diferença nos valores de latência *Commune x RMI* seja alta, podemos concluir que o Commune não inviabiliza o desenvolvimento de sistemas distribuídos neste contexto, pois a latência média de $1256ms$ não é proibitiva para o tipo de sistema estudado neste trabalho, como por exemplo, o OurGrid e o NodeWiz.

Além disso, é preciso considerar que a conexão RMI é síncrona e é composta por apenas uma conexão TCP direta entre os nós. No caso do Commune, a conexão é assíncrona e envolve três conexões TCP.

5.4 Testes

A implementação do Commune foi verificada através de testes automáticos de unidade e de sistema. Além disso, a avaliação mais abrangente do Commune ocorre desde 2 de Fevereiro de 2010 com o lançamento do OurGrid 4.2.0, que possui todos os mecanismos de segurança e o protocolo de conexão habilitados. Nessa versão, o Commune se mostrou estável em ambiente de produção, de modo que a comunidade OurGrid está com cerca de 300 computadores disponíveis no momento da escrita deste trabalho, como pode ser visto no endereço *http://status.ourgrid.org/*.

Capítulo 6

Conclusão

A primeira contribuição deste trabalho foi a identificação de sete requisitos não-funcionais de um sistema de *middleware* para sistemas distribuídos P2P em Java cujos nós precisam manter muitas conexões ativas. Esses requisitos podem ser entendidos como medidas qualitativas para esse tipo de sistema de *middleware*, favorecendo a comparação entre diferentes tecnologias existentes de maneira mais precisa e uma melhor avaliação das mesmas.

Levando em consideração os requisitos elicitados, efetuamos um estudo comparativo dos principais sistemas de *middleware* existentes segundo os modelos de objetos distribuídos, MOMs, e híbrido, para a plataforma Java. Observamos que nenhuma das tecnologias avaliadas atendeu a todos os requisitos plenamente, motivando-nos para a proposição do Commune.

Realizamos uma revisão bibliográfica para definir os mecanismos que compõem a solução de segurança do Commune. Além disto, propusemos os predicados para uma semântica de detecção de falhas abrangente, que oferece o suporte de falhas por parada dos nós e falhas por omissão decorrentes da perda de mensagens no meio de comunicação.

Nós apresentamos detalhadamente os aspectos de projeto e implementação do mecanismo de segurança do Commune e do seu protocolo de conexão que implementa a semântica de detecção de falhas abrangente. Neste caso, enfocamos tanto os elementos de alto nível da arquitetura quanto os detalhes de implementação do Commune

Provamos que o protocolo de conexão implementado atende a todos os predicados da semântica de falhas abrangente. Em um dos predicados foi necessário o uso da ferramenta de verificação formal *Spin*. Realizamos ainda, uma avaliação de viabilidade da implementação

do Commune, mostrando que embora o desempenho desse sistema de *middleware* seja mais lento que o de Java RMI, o seu uso não se torna inviável para o nicho de sistemas estudado.

Diante desses resultados, concluímos que o Commune é um sistema de *middleware* apropriado para os sistemas distribuídos P2P tratados neste trabalho. De fato, dentre os sistemas de *middleware* avaliados, o Commune mostrou-se o único que atende a todos os requisitos não-funcionais levantados.

Como trabalhos futuros propomos: (i) pesquisas para viabilizar a implementação de nós Commune em outras linguagens de programação e a sua interoperabilidade com os nós feitos em Java, (ii) expansão do modelo de sessão do Commune, permitindo diferenciar as conexões reversas feitas por *callback* e as conexões reversas que originaram pelo interesse mútuo e simultâneo entre dois nós, (iii) análise dinâmica do Commune a fim de identificar gargalos e melhorar seu desempenho.

Bibliografia

[1] ADAMS, C., AND FARRELL, S. RFC 2510: Internet X.509 public key infrastructure certificate management protocols. Request for Comments (RFC) 2510, Network Working Group, Mar. 1999.

[2] ANDREWS, G. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000.

[3] BASU, S., COSTA, L., BRASILEIRO, F., BANERJEE, S., SHARMA, P., AND LEE, S.-J. Nodewiz: Fault-tolerant grid information service. *Peer-to-Peer Networking and Applications* (2009).

[4] BESSANI, A., LUNG, L., ALCHIERI, E. P., AND FRAGA, J. Grouppac 3: Estendendo o FT-CORBA para gerenciamento de replicação ativa. In *Workshop de Testes e Tolerância a Falhas (WTF 2004)* (Gramado, RS, Brasil, 2004).

[5] BRASILEIRO, F., GREVE, F., TRONEL, F., HURFIN, M., AND NARZUL, J.-P. L. Eva: An event-based framework for developing specialized communication protocols. In *Proceedings of the IEEE International Symposium on Network Computing and Applications* (Cambridge, 2001), pp. 108 – 119.

[6] CHANDRA, T. D., AND TOUEG, S. Unreliable failure detectors for reliable distributed systems. *J. ACM 43*, 2 (1996), 225–267.

[7] CIRNE, W., BRASILEIRO, F., ANDRADE, N., COSTA, L., ANDRADE, A., NOVAES, R., AND MOWBRAY, M. Labs of the world, unite!!! *Journal of Grid Computing 4*, 3 (2006), 225–246.

[8] CLARKE, E. M., AND EMERSON, E. A. Synthesis of synchronization skeletons for branching time temporal logic. In *Proc. Workshop on Logic of Programs* (Yorktown Heights, NY, 1981), vol. 131, Springer.

[9] COULOURIS, G., AND DOLLIMORE, J. *Distributed systems: concepts and design*. Addison-Wesley Longman Publishing Co., Boston, MA, USA, 2005.

[10] EDG JAVA SECURITY. Virtual organization membership service, http://edg-wp2.web.cern.ch/edg-wp2/security/voms/voms.html, 2003.

[11] ELLISON, C., FRANTZ, B., LAMPSON, B., RIVEST, R., THOMAS, B., AND YLONEN., T. RFC 2693: SPKI certificate theory. Request for Comments (RFC) 2693, Network Working Group, Sept. 1999.

[12] FAIRTHORNE, B. OMG white paper on security. OMG document number 1994/94-04-16, OMG Security Working Group, 1994.

[13] FETTIG, A. *Twisted Network Programming Essentials*. O'Reilly Media, Inc., 2005.

[14] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley, Boston, MA, 1995.

[15] GOKHALE, A., AND SCHMIDT, D. C. The performance of the CORBA dynamic invocation interface and dynamic skeleton over high-speed ATM networks. In *Proceedings of GLOBECOM '96* (London, England, 1996), pp. 50 – 56.

[16] GOSLING, J., JOY, B., AND STEELE, G. L. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[17] HENNING, M. The rise and fall of CORBA. *Queue 4*, 5 (2006), 28–34.

[18] HOLZMANN, G. J. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[19] IOSIF, R. The promela language, http://www.dai-arc.polito.it/dai-arc/manual/tools/jcat/main/node168.html, 1998.

[20] KAVEH, N., AND EMMERICH, W. Deadlock detection in distributed object systems. *SIGSOFT Sotfw. Eng. Notes 26*, 5 (2001), 44–51.

[21] KOHL, J. T., AND NEUMAN, B. C. The Kerberos network authentication service (V5), citeseer.ist.psu.edu/article/kohl93kerberos.html. Tech. Rep. 1510, 1993.

[22] LALISKI, B. RFC 2315: PKCS #7: Cryptographic message syntax. Request for Comments (RFC) 2315, Network Working Group, Mar. 1998.

[23] LIMA, A. Combinando objetos distribuídos e arquiteturas orientadas a eventos em uma infra-estrutura de comunicação para sistemas distribuídos. Master's thesis, UFCG, 2006.

[24] LIMA, A., CIRNE, W., BRASILEIRO, F., AND FIREMAN, D. A case for event-driven distributed objects. In *8th International Symposium on Distributed Objects and Applications (DOA)* (Berlin / Heidelberg, 2006), Springer, pp. 1705–1721.

[25] MCLEAN, S., NAFTEL, J., AND WILLIAMS, K. *Microsoft .NET Remoting*. Microsoft Press, 2002.

[26] OMG. CORBA chapters. `http://www.omg.org/technology/documents/formal/corba_2.htm`.

[27] OMG. History of CORBA. `http://www.omg.org/gettingstarted/history_of_corba.htm`.

[28] OMG. Corba 3.0.3, common object request broker architecture (core specification), 2004.

[29] ORAM, A. *Peer-to-peer - Harnessing the Power of Disruptive Technologies: Peer-to-Peer Models Through the History of the Internet*. O'Reilly Media, 2001, ch. 1, pp. 3–20.

[30] PGP CORPORATION. *An Introduction to Cryptography*, 1998. http://www.xmpp.org.

[31] POSTEL, J. RFC 793: Transmission control protocol. Request for Comments (RFC) 793, Information Sciences Institute - University of Southern California, Sept. 1981.

[32] REESE, G. *Database Programming with JDBC and Java: Distributed Application Architecture*. O'Reilly Media, 2000, ch. 7, pp. 126–145.

[33] SAINT-ANDRE, P. RFC 3920: Extensible messaging and presence protocol (XMPP): Core. Request for Comments (RFC) 3920, Jabber Software Foundation, Oct. 2004.

[34] SCHMIDT, D. C., ROHNERT, H., STAL, M., AND SCHULTZ, D. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Network Objects*. Jonh Wiley & Sons, 2000.

[35] SIMPSON, S. PGP DH vs. RSA FAQ, http://www.scramdisk.clara.net/pgpfaq.html, 1999.

[36] W3C WORKING GROUP. Web services architecture, http://www.w3.org/tr/ws-arch/, 2004.

[37] WOLLRATH, A., RIGGS, R., AND WALDO, J. A distributed object model for the java system. In *Conference on Object-Oriented Technologies* (Toronto, Canada, 1996), pp. 219–232.

[38] XMPP STANDARDS FOUNDATION. Extensible messaging and presence protocol (XMPP), http://www.xmpp.org, 2007.

[39] ZAGHAL, R. Y., AND KHAN, J. I. EFSM/SDL modeling of the original TCP standard (RFC793) and the Congestion Control Mechanism of TCP Reno. Technical Report TR2005-07-22, Internetworking and Media Communications Research Laboratories, Department of Computer Science, Kent State University, Mar. 2005.

[40] ZÖBEL, D. The deadlock problem: a classifying bibliography. *SIGOPS Oper. Syst. Rev. 17*, 4 (1983), 6–15.

[41] ØHRSTRØM, P., AND HASLE, P. *Temporal Logic: From Ancient Ideas to Artificial Intelligence*. Kluwer Academic Publishers, Dordrecht, Boston and London, 1995.

Apêndice A

Modelo Promela de uma conexão Commune

```
1  /* Connection protocol */
2
3  /* The definitions below can be replaced by command line arguments of the
4  spin command
5  */
6  #define A_STATE 0      /* Node A state */
7  #define B_STATE 0      /* Node B state */
8  #define A2B_STATE 0    /* Message in the main channel from A to B */
9  #define B2Ar_STATE 0   /* Message in the reverse channel from B to A */
10 #define B2A_STATE 0    /* Message in the main channel from B to A */
11 #define A2Br_STATE 0   /* Message in the reverse channel from A to B */
12
13 mtype = {
14
15 /* Message types
16 */
17   ri,      /* Register interest(address) */
18   rel,     /* Release(stub) */
19   hb,      /* heartbeat(session, sequence) */
20   us,      /* update status (session, status)*/
21   to,      /* timeout(stub) */
22   msg,     /* message(session, sequence) */
23   msgcb,   /* message with callback (session, sequence) */
```

```
24    nr,       /* notify recovery(stub) */
25    nf,       /* notify failure(stub) */
26
27  /* Message content types
28  */
29    SES,      /* Current session number */
30    N_SES,    /* Other session number */
31    ZERO,     /* Sequence number equals zero */
32    SEQ,      /* Current sequence number */
33    N_SEQ,    /* Other sequence number */
34    ppSEQ,    /* Next sequence number */
35    N_ppSEQ,  /* Sequence number not equals the next */
36
37  /* Node's status types
38  */
39    e,        /* Sender, Receiver - Empty state */
40    down,     /* Sender - The stub is DOWN */
41    ding,     /* Sender - DOWNING - The notify message is in the queue */
42    up,       /* Sender - The stub is DOWN */
43    ui,       /* Sender - UPING - The notify message is in the queue */
44    r0,       /* Receiver - The reverse connection seq number is zero */
45    rs        /* Receiver - The reverse connection seq number is not zero */
46  };
47
48
49  /* Channels
50  */
51  /* Main connection A to B (ses, seq) */
52  chan A2B        = [1] of {mtype, mtype, mtype};
53  /* Reverse connection B to A (ses, status) */
54  chan B2Ar       = [1] of {mtype, mtype, mtype};
55  /* Main connection B to A (ses, seq) */
56  chan B2A        = [1] of {mtype, mtype, mtype};
57  /* Reverse connection A to B (ses, status) */
58  chan A2Br       = [1] of {mtype, mtype, mtype};
59
60  /* This method is invoked to change the node to another state. The number
```

```
61 parameter j represents next node state.
62 */
63 inline jump(j)
64 {
65     if
66     :: (j == 0) -> goto Empty_Empty;
67     :: (j == 1) -> goto Empty_Zero;
68     :: (j == 2) -> goto Empty_GTZero;
69     :: (j == 3) -> goto Down_Empty;
70     :: (j == 4) -> goto Down_Zero;
71     :: (j == 5) -> goto Down_GTZero;
72     :: (j == 6) -> goto Downing_Empty;
73     :: (j == 7) -> goto Up_Empty;
74     :: (j == 8) -> goto Up_Zero;
75     :: (j == 9) -> goto Up_GTZero;
76     :: (j == 10) -> goto Uping_Empty;
77     :: (j == 11) -> goto Uping_Zero;
78     :: (j == 12) -> goto Uping_GTZero;
79     fi;
80 }
81
82 /* This method fills an output channel with a message. The number
83 parameter j represents the message type and the channel parameter
84 outChan is the channel to be filled.
85 */
86 inline fillOutChan(a,outChan)
87 {
88     if
89     :: (a == 0) -> skip;
90     :: (a == 1) -> outChan!hb(SES,ZERO);
91     :: (a == 2) -> outChan!hb(SES,SEQ);
92     :: (a == 3) -> outChan!hb(SES,N_SEQ);
93     :: (a == 4) -> outChan!hb(N_SES,ZERO);
94     :: (a == 5) -> outChan!hb(N_SES,SEQ);
95     :: (a == 6) -> outChan!hb(N_SES,N_SEQ);
96     :: (a == 7) -> outChan!msg(SES,ppSEQ);
97     :: (a == 8) -> outChan!msg(SES,N_ppSEQ);
```

```
98      :: (a == 9) -> outChan!msgcb(SES,ppSEQ);
99      :: (a == 10) -> outChan!msgcb(SES,N_ppSEQ);
100     :: (a == 11) -> outChan!msg(N_SES,SEQ);
101     :: (a == 12) -> outChan!msgcb(N_SES,SEQ);
102     fi;
103 }
104
105 /* This method fills an input channel with a message. The number
106 parameter j represents the message type and the channel parameter
107 inChan is the channel to be filled.
108 */
109 inline fillInRev(b,inRev)
110 {
111     if
112     :: (b == 0) -> skip;
113     :: (b == 1) -> inRev!us(SES,up);
114     :: (b == 2) -> inRev!us(SES,down);
115     :: (b == 3) -> inRev!us(N_SES,up);
116     fi;
117 }
118
119
120 /* This proctype models the behavior of a Commune node.
121 If i == 0, this node is the A node. Otherwise, it is the B node.
122 Each has two main channels (output and input) and two reverse channes
123 (output and input). To see the initialization of the nodes, go to the
124 init clause in the end of this file.
125 */
126 proctype Node(int i; chan outChan; chan outRev; chan inChan; chan inRev)
127 {
128     xs outChan; /* Exclusive write in the output main channel */
129     xr outRev;  /* Exclusive read from the output reverse channel */
130     xr inChan;  /* Exclusive read from the input main channel */
131     xs inRev;   /* Exclusive write in the input reverse channel */
132
133     if
134     :: atomic{ (i == 0) ->
```

```
135             fillOutChan(A2B_STATE,outChan);
136             fillInRev(B2Ar_STATE,inRev);
137             jump(A_STATE)};
138
139         :: atomic{ (i == 1) ->
140             fillOutChan(B2A_STATE,outChan);
141             fillInRev(A2Br_STATE,inRev);
142             jump(B_STATE)};
143         fi;
144
145 Empty_Empty: /* out = EMPTY, in = EMPTY */
146         do
147         :: atomic{jump(3)}; /* Register interest */
148
149         :: atomic{inChan?hb(SES,ZERO); if :: nfull(inRev) ->
150             inRev!us(SES,up); jump(1) :: full(inRev) ->
151                 jump(1) fi}; /* Heartbeat(session, zero) */
152         :: atomic{inChan?hb(SES,ZERO); if :: nfull(inRev) ->
153             inRev!us(SES,down) :: full(inRev) -> skip fi};
154         :: atomic{inChan?hb(SES,ZERO); if :: nfull(inRev) ->
155             inRev!us(N_SES,up) :: full(inRev) -> skip fi}; /* Lost message */
156
157         :: atomic{inChan?hb(SES,SEQ)};
158         :: atomic{inChan?hb(SES,N_SEQ)};
159         :: atomic{inChan?hb(N_SES,ZERO)};
160         :: atomic{inChan?hb(N_SES,SEQ)};
161         :: atomic{inChan?hb(N_SES,N_SEQ)};
162         :: atomic{outRev?us(SES,up)};
163         :: atomic{outRev?us(SES,down)};
164         :: atomic{outRev?us(N_SES,_)};
165         :: atomic{inChan?msg(SES,ppSEQ)};
166         :: atomic{inChan?msg(SES,N_ppSEQ)};
167         :: atomic{inChan?msgcb(SES,ppSEQ)};
168         :: atomic{inChan?msgcb(SES,N_ppSEQ)};
169         :: atomic{inChan?msg(N_SES,_)};
170         :: atomic{inChan?msgcb(N_SES,_)};
171         od;
```

```
172
173 Empty_Zero: /* out = EMPTY, in = R(0) */
174     do
175     :: atomic{jump(4)}; /* Register interest */
176
177     :: atomic{inChan?hb(SES,ZERO); if :: nfull(inRev) ->
178         inRev!us(SES,up) :: full(inRev) -> skip fi};
179     :: atomic{inChan?hb(SES,ZERO); if :: nfull(inRev) ->
180         inRev!us(SES,down) :: full(inRev) -> skip fi};
181     :: atomic{inChan?hb(SES,ZERO); if :: nfull(inRev) ->
182         inRev!us(N_SES,up) fi}; /* Lost message */
183
184     :: atomic{inChan?hb(SES,SEQ) -> jump(0)};
185     :: atomic{inChan?hb(SES,N_SEQ) -> jump(0)};
186     :: atomic{inChan?hb(N_SES,_) -> jump(0)};
187
188     :: atomic{outRev?us(SES,up) -> jump(0)};
189     :: atomic{outRev?us(SES,down) -> jump(0)};
190     :: atomic{outRev?us(N_SES,_) -> jump(0)};
191
192     :: atomic{inChan?msg(SES,ppSEQ) -> jump(2)};
193     :: atomic{inChan?msgcb(SES,ppSEQ) -> jump(9)};
194     :: atomic{inChan?msg(SES,N_ppSEQ) -> jump(0)};
195     :: atomic{inChan?msgcb(SES,N_ppSEQ) -> jump(0)};
196     :: atomic{inChan?msg(N_SES,_) -> jump(0)};
197     :: atomic{inChan?msgcb(N_SES,_) -> jump(0)};
198     od;
199
200 Empty_GTZero: /* out = EMPTY, in = R(S) */
201     do
202     :: atomic{jump(5)}; /* Register interest */
203
204     :: atomic{inChan?hb(SES,ZERO) -> jump(0)};
205
206     :: atomic{inChan?hb(SES,SEQ); if :: nfull(inRev) ->
207         inRev!us(SES,up) :: full(inRev) -> skip fi};
208     :: atomic{inChan?hb(SES,SEQ); if :: nfull(inRev) ->
```

```
209             inRev!us(SES,down) :: full(inRev) -> skip fi};
210         :: atomic{inChan?hb(SES,SEQ); if :: nfull(inRev) ->
211             inRev!us(N_SES,up) :: full(inRev) -> skip fi}; /* Lost message */
212
213         :: atomic{inChan?hb(SES,N_SEQ) -> jump(0)};
214
215         :: atomic{inChan?hb(N_SES,_) -> jump(0)};
216         :: atomic{outRev?us(SES,up) -> jump(0)};
217         :: atomic{outRev?us(SES,down) -> jump(0)};
218         :: atomic{outRev?us(N_SES,_) -> jump(0)};
219         :: atomic{inChan?msg(SES,ppSEQ)};
220         :: atomic{inChan?msg(SES,N_ppSEQ) -> jump(0)};
221         :: atomic{inChan?msgcb(SES,ppSEQ) -> jump(9)};
222         :: atomic{inChan?msgcb(SES,N_ppSEQ) -> jump(0)};
223         :: atomic{inChan?msg(N_SES,_) -> jump(0)};
224         :: atomic{inChan?msgcb(N_SES,_) -> jump(0)};
225         od;
226
227 Down_Empty: /* out = DOWN, in = EMPTY */
228         do
229         :: atomic{jump(0)}; /* Release */
230
231         :: atomic{inChan?hb(SES,ZERO); if :: nfull(inRev) ->
232             inRev!us(SES,up); jump(4) :: full(inRev) -> jump(4) fi};
233         :: atomic{inChan?hb(SES,ZERO); if :: nfull(inRev) ->
234             inRev!us(SES,down); jump(4) :: full(inRev) -> jump(4) fi};
235         :: atomic{inChan?hb(SES,ZERO); if :: nfull(inRev) ->
236             inRev!us(N_SES,up); jump(4) :: full(inRev) ->
237                 jump(4) fi}; /* Lost message */
238
239         :: atomic{inChan?hb(SES,SEQ)};
240         :: atomic{inChan?hb(SES,N_SEQ)};
241
242         :: atomic{inChan?hb(N_SES,_)};
243         :: atomic{outRev?us(SES,up) -> jump(10)};
244         :: atomic{outRev?us(SES,down)};
245         :: atomic{outRev?us(N_SES,_)};
```

```
246     :: atomic{inChan?msg(SES,ppSEQ)};
247     :: atomic{inChan?msg(SES,N_ppSEQ)};
248     :: atomic{inChan?msgcb(SES,ppSEQ)};
249     :: atomic{inChan?msgcb(SES,N_ppSEQ)};
250     :: atomic{inChan?msg(N_SES,_)};
251     :: atomic{inChan?msgcb(N_SES,_)};
252
253     :: atomic{nfull(outChan) -> outChan!hb(SES,ZERO)};
254     :: atomic{nfull(outChan) ->
255         outChan!hb(SES,N_SEQ)}; /* Lost message */
256     :: atomic{nfull(outChan) ->
257         outChan!hb(N_SES,ZERO)}; /* Lost message */
258     :: atomic{nfull(outChan) ->
259         outChan!hb(N_SES,SEQ)}; /* Lost message */
260     :: atomic{nfull(outChan) ->
261         outChan!hb(N_SES,N_SEQ)}; /* Lost message */
262   od;
263
264 Down_Zero: /* out = DOWN, in = R(0) */
265   do
266     :: atomic{jump(0)}; /* Release */
267
268     :: atomic{inChan?hb(SES,ZERO); if :: nfull(inRev) ->
269         inRev!us(SES,up) :: full(inRev) -> skip fi};
270     :: atomic{inChan?hb(SES,ZERO); if :: nfull(inRev) ->
271         inRev!us(SES,down) :: full(inRev) -> skip fi};
272     :: atomic{inChan?hb(SES,ZERO); if :: nfull(inRev) ->
273         inRev!us(N_SES,up) :: full(inRev) -> skip fi}; /* Lost message */
274
275     :: atomic{inChan?hb(SES,SEQ) -> jump(3)};
276     :: atomic{inChan?hb(SES,N_SEQ) -> jump(3)};
277     :: atomic{inChan?hb(N_SES,_) -> jump(3)};
278
279     :: atomic{outRev?us(SES,up) -> jump(11)};
280     :: atomic{outRev?us(SES,down) -> jump(3)};
281     :: atomic{outRev?us(N_SES,_) -> jump(3)};
282     :: atomic{jump(3)}; /* Timeout */
```

```
283       :: atomic{inChan?msg(SES,ppSEQ) -> jump(5)};
284       :: atomic{inChan?msg(SES,N_ppSEQ) -> jump(3)};
285       :: atomic{inChan?msgcb(SES,ppSEQ) -> jump(9)};
286       :: atomic{inChan?msgcb(SES,N_ppSEQ) -> jump(3)};
287       :: atomic{inChan?msg(N_SES,_) -> jump(3)};
288       :: atomic{inChan?msgcb(N_SES,_) -> jump(3)};
289
290       :: atomic{nfull(outChan) -> outChan!hb(SES,ZERO)};
291       :: atomic{nfull(outChan) ->
292           outChan!hb(SES,N_SEQ)}; /* Lost message */
293       :: atomic{nfull(outChan) ->
294           outChan!hb(N_SES,ZERO)}; /* Lost message */
295       :: atomic{nfull(outChan) ->
296           outChan!hb(N_SES,SEQ)}; /* Lost message */
297       :: atomic{nfull(outChan) ->
298           outChan!hb(N_SES,N_SEQ)}; /* Lost message */
299     od;
300
301 Down_GTZero: /* out = DOWN, in = R(S) */
302     do
303       :: atomic{jump(0)}; /* Release */
304       :: atomic{inChan?hb(SES,ZERO) -> jump(3)};
305
306       :: atomic{inChan?hb(SES,SEQ); if :: nfull(inRev) ->
307           inRev!us(SES,up):: full(inRev) -> skip fi};
308       :: atomic{inChan?hb(SES,SEQ); if :: nfull(inRev) ->
309           inRev!us(SES,down) :: full(inRev) -> skip fi};
310       :: atomic{inChan?hb(SES,SEQ); if :: nfull(inRev) ->
311           inRev!us(N_SES,up) :: full(inRev) -> skip fi}; /* Lost message */
312
313       :: atomic{inChan?hb(SES,N_SEQ) -> jump(3)};
314
315       :: atomic{inChan?hb(N_SES,_) -> jump(3)};
316       :: atomic{outRev?us(SES,up) -> jump(12)};
317       :: atomic{outRev?us(SES,down) -> jump(3)};
318       :: atomic{outRev?us(N_SES,_) -> jump(3)};
319       :: atomic{jump(3)}; /* Timeout */
```

```
320       :: atomic{inChan?msg(SES,ppSEQ)};
321       :: atomic{inChan?msg(SES,N_ppSEQ) -> jump(3)};
322       :: atomic{inChan?msgcb(SES,ppSEQ) -> jump(9)};
323       :: atomic{inChan?msgcb(SES,N_ppSEQ) -> jump(3)};
324       :: atomic{inChan?msg(N_SES,_) -> jump(3)};
325       :: atomic{inChan?msgcb(N_SES,_) -> jump(3)};
326
327       :: atomic{nfull(outChan) -> outChan!hb(SES,ZERO)};
328       :: atomic{nfull(outChan) ->
329           outChan!hb(SES,N_SEQ)}; /* Lost message */
330       :: atomic{nfull(outChan) ->
331           outChan!hb(N_SES,ZERO)}; /* Lost message */
332       :: atomic{nfull(outChan) ->
333           outChan!hb(N_SES,SEQ)}; /* Lost message */
334       :: atomic{nfull(outChan) ->
335           outChan!hb(N_SES,N_SEQ)}; /* Lost message */
336     od;
337
338 Downing_Empty: /* out = DOWNING, in = EMPTY */
339     do
340       :: atomic{jump(0)}; /* Release */
341
342       :: atomic{inChan?hb(SES,ZERO)};
343       :: atomic{inChan?hb(SES,SEQ)};
344       :: atomic{inChan?hb(SES,N_SEQ)};
345       :: atomic{inChan?hb(N_SES,ZERO)};
346       :: atomic{inChan?hb(N_SES,SEQ)};
347       :: atomic{inChan?hb(N_SES,N_SEQ)};
348       :: atomic{outRev?us(SES,up)};
349       :: atomic{outRev?us(SES,down)};
350       :: atomic{outRev?us(N_SES,_)};
351       :: atomic{inChan?msg(SES,ppSEQ)};
352       :: atomic{inChan?msg(SES,N_ppSEQ)};
353       :: atomic{inChan?msgcb(SES,ppSEQ)};
354       :: atomic{inChan?msgcb(SES,N_ppSEQ)};
355       :: atomic{inChan?msg(N_SES,_)};
356       :: atomic{inChan?msgcb(N_SES,_)};
```

```
357
358     :: atomic{jump(3)}; /* Notify failure */
359     od;
360
361 Up_Empty: /* out = UP, in = EMPTY */
362     do
363     :: atomic{jump(0)}; /* Release */
364
365     :: atomic{inChan?hb(SES,ZERO); if :: nfull(inRev) ->
366         inRev!us(SES,up); jump(8) :: full(inRev) -> jump(8) fi};
367     :: atomic{inChan?hb(SES,ZERO); if :: nfull(inRev) ->
368         inRev!us(SES,down); jump(8) :: full(inRev) -> jump(8) fi};
369     :: atomic{inChan?hb(SES,ZERO); if :: nfull(inRev) ->
370         inRev!us(N_SES,up); jump(8) :: full(inRev) ->
371             jump(8) fi}; /* Lost message */
372
373     :: atomic{inChan?hb(SES,SEQ) -> jump(6)};
374     :: atomic{inChan?hb(SES,N_SEQ) -> jump(6)};
375
376     :: atomic{inChan?hb(N_SES,_) -> jump(6)};
377
378     :: atomic{outRev?us(SES,up)};
379     :: atomic{outRev?us(SES,down) -> jump(6)};
380     :: atomic{outRev?us(N_SES,_) -> jump(6)};
381     :: atomic{jump(6)}; /* Timeout */
382     :: atomic{inChan?msg(SES,ppSEQ) -> jump(6)};
383     :: atomic{inChan?msg(SES,N_ppSEQ) -> jump(6)};
384     :: atomic{inChan?msgcb(SES,ppSEQ) -> jump(6)};
385     :: atomic{inChan?msgcb(SES,N_ppSEQ) -> jump(6)};
386     :: atomic{inChan?msg(N_SES,_) -> jump(6)};
387     :: atomic{inChan?msgcb(N_SES,_) -> jump(6)};
388
389     :: atomic{nfull(outChan) -> outChan!hb(SES,ZERO)};
390     :: atomic{nfull(outChan) ->
391         outChan!hb(N_SES,ZERO)}; /* Lost message */
392     :: atomic{nfull(outChan) -> outChan!hb(SES,SEQ)};
393     :: atomic{nfull(outChan) ->
```

```
394             outChan!hb(SES,N_SEQ)}; /* Lost message */
395         :: atomic{nfull(outChan) ->
396             outChan!hb(N_SES,SEQ)}; /* Lost message */
397         :: atomic{nfull(outChan) ->
398             outChan!hb(N_SES,N_SEQ)}; /* Lost message */
399         :: atomic{nfull(outChan) -> outChan!msg(SES,ppSEQ)};
400         :: atomic{nfull(outChan) ->
401             outChan!msg(SES,N_ppSEQ)}; /* Lost message */
402         :: atomic{nfull(outChan) -> outChan!msgcb(SES,ppSEQ)};
403         :: atomic{nfull(outChan) ->
404             outChan!msgcb(SES,N_ppSEQ)}; /* Lost message */
405         :: atomic{nfull(outChan) ->
406             outChan!msg(N_SES,ppSEQ)}; /* Lost message */
407         :: atomic{nfull(outChan) ->
408             outChan!msg(N_SES,N_ppSEQ)}; /* Lost message */
409         :: atomic{nfull(outChan) ->
410             outChan!msgcb(N_SES,ppSEQ)}; /* Lost message */
411         :: atomic{nfull(outChan) ->
412             outChan!msgcb(N_SES,N_ppSEQ)}; /* Lost message */
413         od;
414
415     Up_Zero: /* out = UP, in = R(0) */
416         do
417         :: atomic{jump(0)}; /* Release */
418
419         :: atomic{inChan?hb(SES,ZERO); if :: nfull(inRev) ->
420             inRev!us(SES,up) :: full(inRev) -> skip fi};
421         :: atomic{inChan?hb(SES,ZERO); if :: nfull(inRev) ->
422             inRev!us(SES,down) :: full(inRev) -> skip fi};
423         :: atomic{inChan?hb(SES,ZERO); if :: nfull(inRev) ->
424             inRev!us(N_SES,up) :: full(inRev) -> skip fi}; /* Lost message */
425
426         :: atomic{inChan?hb(SES,SEQ) -> jump(6)};
427         :: atomic{inChan?hb(SES,N_SEQ) -> jump(6)};
428         :: atomic{inChan?hb(N_SES,_) -> jump(6)};
429         :: atomic{outRev?us(SES,up)};
430         :: atomic{outRev?us(SES,down) -> jump(6)};
```

```
431       :: atomic{outRev?us(N_SES,_) -> jump(6)};
432       :: atomic{jump(6)}; /* Timeout */
433       :: atomic{inChan?msg(SES,ppSEQ) -> jump(9)};
434       :: atomic{inChan?msg(SES,N_ppSEQ) -> jump(6)};
435       :: atomic{inChan?msgcb(SES,ppSEQ) -> jump(9)};
436       :: atomic{inChan?msgcb(SES,N_ppSEQ) -> jump(6)};
437       :: atomic{inChan?msg(N_SES,_) -> jump(6)};
438       :: atomic{inChan?msgcb(N_SES,_) -> jump(6)};
439
440       :: atomic{nfull(outChan) -> outChan!hb(SES,ZERO)};
441       :: atomic{nfull(outChan) ->
442           outChan!hb(N_SES,ZERO)}; /* Lost message */
443       :: atomic{nfull(outChan) -> outChan!hb(SES,SEQ)};
444       :: atomic{nfull(outChan) ->
445           outChan!hb(SES,N_SEQ)}; /* Lost message */
446       :: atomic{nfull(outChan) ->
447           outChan!hb(N_SES,SEQ)}; /* Lost message */
448       :: atomic{nfull(outChan) ->
449           outChan!hb(N_SES,N_SEQ)}; /* Lost message */
450       :: atomic{nfull(outChan) -> outChan!msg(SES,ppSEQ)};
451       :: atomic{nfull(outChan) ->
452           outChan!msg(SES,N_ppSEQ)}; /* Lost message */
453       :: atomic{nfull(outChan) -> outChan!msgcb(SES,ppSEQ)};
454       :: atomic{nfull(outChan) ->
455           outChan!msgcb(SES,N_ppSEQ)}; /* Lost message */
456       :: atomic{nfull(outChan) ->
457           outChan!msg(N_SES,ppSEQ)}; /* Lost message */
458       :: atomic{nfull(outChan) ->
459           outChan!msg(N_SES,N_ppSEQ)}; /* Lost message */
460       :: atomic{nfull(outChan) ->
461           outChan!msgcb(N_SES,ppSEQ)}; /* Lost message */
462       :: atomic{nfull(outChan) ->
463           outChan!msgcb(N_SES,N_ppSEQ)}; /* Lost message */
464     od;
465
466 Up_GTZero: /* out = UP, in = R(S) */
467     do
```

```
468     :: atomic{jump(0)}; /* Release */
469     :: atomic{inChan?hb(SES,ZERO) -> jump(6)};
470
471     :: atomic{inChan?hb(SES,SEQ); if :: nfull(inRev) ->
472         inRev!us(SES,up) :: full(inRev) -> skip fi};
473     :: atomic{inChan?hb(SES,SEQ); if :: nfull(inRev) ->
474         inRev!us(SES,down) :: full(inRev) -> skip fi};
475     :: atomic{inChan?hb(SES,SEQ); if :: nfull(inRev) ->
476         inRev!us(N_SES,up) :: full(inRev) -> skip fi}; /* Lost message */
477
478     :: atomic{inChan?hb(SES,N_SEQ) -> jump(6)};
479     :: atomic{inChan?hb(N_SES,_) -> jump(6)};
480     :: atomic{outRev?us(SES,up)};
481     :: atomic{outRev?us(SES,down) -> jump(6)};
482     :: atomic{outRev?us(N_SES,_) -> jump(6)};
483     :: atomic{jump(6)}; /* Timeout */
484     :: atomic{inChan?msg(SES,ppSEQ)};
485     :: atomic{inChan?msg(SES,N_ppSEQ) -> jump(6)};
486     :: atomic{inChan?msgcb(SES,ppSEQ)};
487     :: atomic{inChan?msgcb(SES,N_ppSEQ) -> jump(6)};
488     :: atomic{inChan?msg(N_SES,_) -> jump(6)};
489     :: atomic{inChan?msgcb(N_SES,_) -> jump(6)};
490
491     :: atomic{nfull(outChan) -> outChan!hb(SES,ZERO)};
492     :: atomic{nfull(outChan) ->
493         outChan!hb(N_SES,ZERO)}; /* Lost message */
494     :: atomic{nfull(outChan) -> outChan!hb(SES,SEQ)};
495     :: atomic{nfull(outChan) ->
496         outChan!hb(SES,N_SEQ)}; /* Lost message */
497     :: atomic{nfull(outChan) ->
498         outChan!hb(N_SES,SEQ)}; /* Lost message */
499     :: atomic{nfull(outChan) ->
500         outChan!hb(N_SES,N_SEQ)}; /* Lost message */
501     :: atomic{nfull(outChan) -> outChan!msg(SES,ppSEQ)};
502     :: atomic{nfull(outChan) ->
503         outChan!msg(SES,N_ppSEQ)}; /* Lost message */
504     :: atomic{nfull(outChan) -> outChan!msgcb(SES,ppSEQ)};
```

```
505     :: atomic{nfull(outChan) ->
506         outChan!msgcb(SES,N_ppSEQ)}; /* Lost message */
507     :: atomic{nfull(outChan) ->
508         outChan!msg(N_SES,ppSEQ)}; /* Lost message */
509     :: atomic{nfull(outChan) ->
510         outChan!msg(N_SES,N_ppSEQ)}; /* Lost message */
511     :: atomic{nfull(outChan) ->
512         outChan!msgcb(N_SES,ppSEQ)}; /* Lost message */
513     :: atomic{nfull(outChan) ->
514         outChan!msgcb(N_SES,N_ppSEQ)}; /* Lost message */
515     od;
516
517 Uping_Empty: /* out = UPING, in = EMPTY */
518     do
519     :: atomic{jump(0)}; /* Release */
520
521     :: atomic{inChan?hb(SES,ZERO); if :: nfull(inRev) ->
522         inRev!us(SES,up); jump(11) :: full(inRev) -> jump(11) fi};
523     :: atomic{inChan?hb(SES,ZERO); if :: nfull(inRev) ->
524         inRev!us(SES,down); jump(11) :: full(inRev) -> jump(11) fi};
525     :: atomic{inChan?hb(SES,ZERO); if :: nfull(inRev) ->
526         inRev!us(N_SES,up); jump(11) :: full(inRev) -> jump(11) fi};
527             /* Lost message */
528
529     :: atomic{inChan?hb(SES,SEQ) -> jump(6)};
530     :: atomic{inChan?hb(SES,N_SEQ) -> jump(6)};
531
532     :: atomic{inChan?hb(N_SES,_) -> jump(6)};
533     :: atomic{outRev?us(SES,up)};
534     :: atomic{outRev?us(SES,down) -> jump(6)};
535     :: atomic{outRev?us(N_SES,_) -> jump(6)};
536     :: atomic{jump(6)}; /* Timeout */
537     :: atomic{inChan?msg(SES,ppSEQ) -> jump(6)};
538     :: atomic{inChan?msg(SES,N_ppSEQ) -> jump(6)};
539     :: atomic{inChan?msgcb(SES,ppSEQ) -> jump(6)};
540     :: atomic{inChan?msgcb(SES,N_ppSEQ) -> jump(6)};
541     :: atomic{inChan?msg(N_SES,_) -> jump(6)};
```

```
542     :: atomic{inChan?msgcb(N_SES,_) -> jump(6)};
543
544     :: atomic{jump(7)}; /* Notify recovery */
545     :: atomic{nfull(outChan) -> outChan!hb(SES,ZERO)};
546     :: atomic{nfull(outChan) ->
547         outChan!hb(SES,N_SEQ)}; /* Lost message */
548     :: atomic{nfull(outChan) ->
549         outChan!hb(N_SES,ZERO)}; /* Lost message */
550     :: atomic{nfull(outChan) ->
551         outChan!hb(N_SES,SEQ)}; /* Lost message */
552     :: atomic{nfull(outChan) ->
553         outChan!hb(N_SES,N_SEQ)}; /* Lost message */
554     od;
555
556 Uping_Zero: /* out = UPING, in = R(0) */
557     do
558     :: atomic{jump(0)}; /* Release */
559
560     :: atomic{inChan?hb(SES,ZERO); if :: nfull(inRev) ->
561         inRev!us(SES,up) :: full(inRev) -> skip fi};
562     :: atomic{inChan?hb(SES,ZERO); if :: nfull(inRev) ->
563         inRev!us(SES,down) :: full(inRev) -> skip fi};
564     :: atomic{inChan?hb(SES,ZERO); if :: nfull(inRev) ->
565         inRev!us(N_SES,up) :: full(inRev) -> skip fi}; /* Lost message */
566
567     :: atomic{inChan?hb(SES,SEQ) -> jump(6)};
568     :: atomic{inChan?hb(SES,N_SEQ) -> jump(6)};
569     :: atomic{inChan?hb(N_SES,_) -> jump(6)};
570     :: atomic{outRev?us(SES,up)};
571     :: atomic{outRev?us(SES,down) -> jump(6)};
572     :: atomic{outRev?us(N_SES,_) -> jump(6)};
573     :: atomic{jump(6)}; /* Timeout */
574     :: atomic{inChan?msg(SES,ppSEQ) -> jump(12)};
575     :: atomic{inChan?msg(SES,N_ppSEQ) -> jump(6)};
576     :: atomic{inChan?msgcb(SES,ppSEQ) -> jump(9)};
577     :: atomic{inChan?msgcb(SES,N_ppSEQ) -> jump(6)};
578     :: atomic{inChan?msg(N_SES,_) -> jump(6)};
```

```
579     :: atomic{inChan?msgcb(N_SES,_) -> jump(6)};
580
581     :: atomic{jump(8)}; /* Notify recovery */
582     :: atomic{nfull(outChan) -> outChan!hb(SES,ZERO)};
583     :: atomic{nfull(outChan) ->
584         outChan!hb(SES,N_SEQ)}; /* Lost message */
585     :: atomic{nfull(outChan) ->
586         outChan!hb(N_SES,ZERO)}; /* Lost message */
587     :: atomic{nfull(outChan) ->
588         outChan!hb(N_SES,SEQ)}; /* Lost message */
589     :: atomic{nfull(outChan) ->
590         outChan!hb(N_SES,N_SEQ)}; /* Lost message */
591     od;
592
593 Uping_GTZero: /* out = UPING, in = R(S) */
594     do
595     :: atomic{jump(0)}; /* Release */
596     :: atomic{inChan?hb(SES,ZERO) -> jump(6)};
597
598     :: atomic{inChan?hb(SES,SEQ); if :: nfull(inRev) ->
599         inRev!us(SES,up) :: full(inRev) -> skip fi};
600     :: atomic{inChan?hb(SES,SEQ); if :: nfull(inRev) ->
601         inRev!us(SES,down) :: full(inRev) -> skip fi};
602     :: atomic{inChan?hb(SES,SEQ); if :: nfull(inRev) ->
603         inRev!us(N_SES,up) :: full(inRev) -> skip fi}; /* Lost message */
604
605     :: atomic{inChan?hb(SES,N_SEQ) -> jump(6)};
606     :: atomic{inChan?hb(N_SES,_) -> jump(6)};
607     :: atomic{outRev?us(SES,up)};
608     :: atomic{outRev?us(SES,down) -> jump(6)};
609     :: atomic{outRev?us(N_SES,_) -> jump(6)};
610     :: atomic{jump(6)}; /* Timeout */
611     :: atomic{inChan?msg(SES,ppSEQ)};
612     :: atomic{inChan?msg(SES,N_ppSEQ) -> jump(6)};
613     :: atomic{inChan?msgcb(SES,ppSEQ) -> jump(9)};
614     :: atomic{inChan?msgcb(SES,N_ppSEQ) -> jump(6)};
615     :: atomic{inChan?msg(N_SES,_) -> jump(6)};
```

```
616     :: atomic{inChan?msgcb(N_SES,_) -> jump(6)};
617
618     :: atomic{jump(9)}; /* Notify recovery */
619     :: atomic{nfull(outChan) -> outChan!hb(SES,ZERO)};
620     :: atomic{nfull(outChan) ->
621         outChan!hb(SES,N_SEQ)}; /* Lost message */
622     :: atomic{nfull(outChan) ->
623         outChan!hb(N_SES,ZERO)}; /* Lost message */
624     :: atomic{nfull(outChan) ->
625         outChan!hb(N_SES,SEQ)}; /* Lost message */
626     :: atomic{nfull(outChan) ->
627         outChan!hb(N_SES,N_SEQ)}; /* Lost message */
628     od;
629 }
630
631 /* Start the processes */
632 init {
633     atomic {
634         run Node(0, A2B, A2Br, B2A, B2Ar);
635         run Node(1, B2A, B2Ar, A2B, A2Br)
636     }
637 }
638
639 /*
640 Usage instructions:
641
642 for (( a = 0; a <= 12; a++ )) do
643     for (( b = 0 ; b <= 12; b++ )) do
644         for (( c = 0; c <= 13; c++ )) do
645             for (( d = 0; d <= 4; d++ )) do
646                 for (( e = 0; e <= 13; e++ )) do
647                     for (( f = 0; f <= 4; f++ )) do
648                         spin -a -DA_STATE=$a -DB_STATE=$b -DA2B_STATE=$c
                                -DB2Ar_STATE=$d -DB2A_STATE=$e -DA2Br_STATE=$f
                                 conn.pml
649                         gcc -o pan pan.c -DSAFETY -DXUSAFE
650                         ./pan -m100000 > output.$a.$b.$c.$d.$e.$f
```

```
651                                 done
652                             done
653                         done
654                     done
655                 done
656 done
657 */
```