Integrando uma Semântica de Falhas Consistente na Comunicação Assíncrona de Objetos Distribuídos

Rodrigo Miranda[1], **Francisco Brasileiro**[1]

[1]Universidade Federal de Campina Grande
Departamento de Sistemas e Computação
Laboratório de Sistemas Distribuídos
Av. Aprígio Veloso, s/n, Bloco CO
58.109-970, Campina Grande - PB, Brazil

{rvilar,fubica}@dsc.ufcg.edu.br

Abstract. *The current technologies for distributed systems development do not match the needs of complex systems and that maintains several connections opened at the same time in a node. Asynchronous communication support for distributed objects helps the implementation of these systems. In this paper, we define four requirements for a consistent semantic of failures for this kind of communication support. Moreover, we designed and formally verified a Connection Protocol which meets these requirements. We also describe the architecture of a middleware that supports asynchronous distributed objects and uses this Connection Protocol.*

Resumo. *As tecnologias disponíveis para o desenvolvimento sistemas distribuídos não atendem perfeitamente à classe de sistemas complexos e que precisa manter diversas conexões abertas simultaneamente em um nó. A comunicação assíncrona de objetos distribuídos facilita a construção destes sistemas. Neste trabalho, definimos quatro requisitos para uma semântica de falhas consistente para este tipo de comunicação. Além disso, projetamos e verificamos formalmente um Protoloco de Conexão que atende a esses requisitos. Por fim, descrevemos a arquitetura de uma plataforma para objetos distribuídos assíncrona que usa este Protocolo de Conexão.*

1. Introdução

Com a popularização da Internet, muitos sistemas distribuídos desenvolvidos atualmente têm sido implantados na grande rede. Estes sistemas possuem complexidade elevada devido a sua própria natureza distribuída e por causa das características do meio de comunicação utilizado, que incluem a ocorrência de falhas, obstáculos na conectividade da rede e vulnerabilidades de segurança.

Muitas plataformas de comunicação, chamadas de *middleware*, têm surgido com o intuito de facilitar o desenvolvimento de sistemas distribuídos. Elas provêm um *framework* com a implementação de serviços relativos à comunicação remota e à distribuição do sistema. Deste modo, o programador pode focar na implementação da lógica de negócio do sistema.

O projeto de uma plataforma de comunicação se torna melhor quando prevê os requisitos não-funcionais das aplicações que pretende suportar e os implementa automaticamente. Portanto, no contexto dos sistemas distribuídos implantados na Internet,

a plataforma de comunicação deve prover serviços que auxiliem a tolerância a falhas, a sobreposição de obstáculos na conectividade da rede e a implementação de componentes seguros. Além disto, a plataforma de comunicação deve possuir uma interface de programação que não adicione complexidade à estrutura dos componentes distribuídos.

A possibilidade de falhas é um aspecto inerente a qualquer sistema distribuído que é agravado em sistemas implantados na Internet. Se um nó falhar ou ficar inacessível, o sistema pode ficar esperando infinitamente por um serviço que não será feito. Se houver uma perda de mensagem no meio de comunicação, o estado do sistema pode ficar inconsistente e prejudicar o trabalho como um todo. Por estes motivos, a detecção de falhas é uma questão crucial no projeto e na implementação de sistemas distribuídos.

Por causa da limitação dos endereços disponíveis no protocolo IPv4 e por questões de segurança, alguns mecanismos que alteram a conectividade de nós na Internet foram criados e têm sido amplamente utilizados. Os *firewalls* são instalados em *gateways* que se localizam entre uma rede interna e a Internet. Neste cenário, é utilizada a estratégia de fechar todas as portas de comunicação, liberando apenas os serviços que são necessários aos clientes da rede interna. Se preciso, os *firewalls* permitem que alguns computadores da rede interna atuem como servidores. No entanto, os computadores restantes podem atuar apenas como clientes.

Nas redes privadas, apenas o *gateway* possui um IP público e assume um papel de *proxy* para os computadores da rede interna. Desta forma, a conexão direta com um computador da rede privada só é possível se for configurado um redirecionamento de porta no *gateway*.

Os *firewalls* e as redes privadas funcionam bem para sistemas no modelo cliente/servidor, pois neste modelo existe uma distinção clara entre os nós clientes e servidores. Os clientes podem estar em redes privadas e sob a atuação de *firewalls*. Os servidores precisam possuir endereço público e diretamente acessível. O protocolo HTTP é um exemplo deste modelo, onde existem clientes (*browsers*) e servidores *web*.

Todavia, nos últimos anos têm surgido muitos sistemas *peer-to-peer*, nos quais um mesmo nó pode atuar como cliente e servidor simultaneamente. Consequentemente, todos os nós precisariam possuir um endereço público e ser diretamente acessíveis. Entretanto, existem algumas alternativas para sobrepor os obstáculos de conectividade, permitindo que computadores sob *firewalls* ou redes privadas atuem como servidores. Dentre as alternativas viáveis para os sistemas *peer-to-peer*, podemos citar o uso de servidores intermediários, como é feito no envio e recebimento de e-mails, e o tunelamento da conexão sobre HTTP.

Existem diversos mecanismos desenvolvidos a fim de prover segurança para sistemas sobre a Internet, como o uso de chaves criptográficas e de certificação. Porém, a implantação de alguns sistemas distribuídos pode se tornar inviável se for adotada uma solução de segurança com estrutura complexa, como o uso de certificados X.509 assinados por uma autoridade certificadora. A plataforma de comunicação deve prover um portifólio de soluções de segurança flexível, que possa ser configurado facilmente pelo programador. Se for preciso usar alguma solução de segurança, este serviço já deve ser provido pela plataforma. Porém, se não houver requisitos de segurança, a implantação do sistema deve continuar fácil.

As linguagens orientadas a objeto (OO) permitem a modularização de grandes sistemas, dividindo-os em componentes independentes, que interagem através de interfaces bem definidas e pouco mutáveis. Um programador ou uma equipe de desenvolvimento pode trabalhar em um componente, sem se preocupar com os detalhes dos componentes adjacentes, bastando apenas conhecer a sua interface.

O modelo de objetos distribuídos é uma adaptação do paradigma de programação OO para sistemas distribuídos. A troca de mensagens entre os nós é encapsulada pela plataforma de comunicação, de forma que objetos remotos, localizados em diferentes processos, podem se comunicar com invocações de métodos e tendo a impressão que estão no mesmo espaço de endereçamento. Este modelo representa um grande progresso para o desenvolvimento de sistemas distribuídos [Coulouris and Dollimore 2005].

A fim de emular o comportamento de objetos locais, a maior parte das implementações do modelo de objetos distribuídos utiliza uma abordagem síncrona, na qual o fluxo de execução (*Thread*) no processo do cliente[1] 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. Como exemplo de tecnologias que utilizaram a abordagem síncrona, pode-se citar *Java RMI* [Wollrath et al. 1996] e *CORBA* [OMG 2004].

Todavia, esta abordagem síncrona pode não ser ideal para alguns sistemas distribuídos devido a dois fatores: (i) o desempenho e a escalabilidade dos componentes que mantêm muitas conexões abertas simultaneamente pode ser prejudicado; e (ii) se algum componente do sistema distribuído precisar efetuar alguma atividade enquanto espera pela resposta da invocação remota, ele será obrigado a utilizar uma programação com múltiplos fluxos, o que reduz o encapsulamento dos objetos. Isto ocorre porque os objetos precisarão conhecer os detalhes internos uns dos outros, a fim de sincronizar corretamente o estado em memória para os diversos fluxos ativos.

Uma solução alternativa para a programação de sistemas distribuídos é a abordagem assíncrona, na qual existe uma fila de mensagens ou eventos, em cada processo do sistema [Schmidt et al. 2000]. Estas mensagens podem ser consumidas e executadas pela aplicação sem concorrência. Logo, é possível preservar o encapsulamento e aumentar a escalabilidade do sistema. A desvantagem desta abordagem é a pouca integração com as linguagens OO. Ao invés de invocar métodos de objetos, o programador precisa usar primitivas de baixo nível (*send/receive*). Além disto, estas primitivas não permitem a checagem de tipo nas interações remotas. Portanto muitos dos erros que seriam detectados em tempo de compilação, aparecerão apenas em tempo de execução.

Para o desenvolvimento de sistemas distribuídos que precisam manter muitas conexões abertas simultaneamente em algum nó, seria ideal que a plataforma de comunicação combinasse as abordagens síncrona e assíncrona. A interface OO contribuiria para a modularização do sistema e o assincronismo evitaria a explosão do número de fluxos de execução, melhorando a escalabilidade do sistema.

Na seção 2 deste artigo, faremos um levantamento das plataformas disponíveis para o desenvolvimento de sistemas distribuídos. A seção 3 formaliza o problema da ausência de uma semântica de falhas consistente para a comunicação assíncrona de obje-

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

tos distribuídos. Na seção 4, um protocolo de conexão que define uma semântica de falhas consistente será apresentado e validado. A seção 5 demonstra os detalhes de projeto do protocolo de conexão em uma plataforma de comunicação. Por fim, a última seção mostra as conclusões e propõe os trabalhos futuros.

2. Trabalhos relacionados

Uma das primeiras e mais utilizadas tecnologias de objetos distribuídos foi o *Java RMI* [Wollrath et al. 1996], que utiliza a abordagem síncrona para a comunicação remota entre objetos. O escopo dos fluxos de execução locais pode atingir processos remotos, o que facilita a legibilidade do código de invocação remota e a detecção e o tratamento de falhas, pois permite o uso de estruturas *try-catch*, como ocorre na programação local. Todavia, este escopo distribuído torna a sincronização dos fluxos mais complexa, pois reduz o encapsulamento dos objetos, e dificulta a realização de testes.

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. Na linguagem *Java*, cada fluxo de execução de um processo ocupa no mínimo 256 KB [Sun Microsystems 2009]. Logo, pode haver um estouro de memória quando forem abertos muitas conexões simultâneas. Além disto, *Java RMI* não possui uma abordagem apropriada para o tratamento dos obstáculos na conectividade da Internet, pois exige o acesso direto entre os nós em comunicação. Este requisito só pode ser atendido 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 solução adiciona *overhead* na comunicação.

A Microsoft também criou uma tecnologia para objetos distribuídos, chamada *.NET Remoting* [McLean et al. 2002]. Ela possui características semelhantes ao *Java RMI*, portanto enfrenta as mesmas limitações para o desenvolvimento de sistemas distribuídos sobre a Internet que precisem manter milhares de conexões abertas simultaneamente.

Os *MOMs (Message-Oriented Middlewares)* são plataformas para comunicação em grupo no padrão *publish/subscribe* [Brasileiro et al. 2001]. A publicação e o consumo de eventos usam primitivas de baixo nível semelhantes ao envio (*send*) e recebimento (*receive*) de mensagens, portanto a integração com a linguagem de programação OO não é tão boa quanto em objetos distribuídos. Em relação às falhas do sistema, os MOMs permitem que as mensagens sejam armazenadas pela plataforma quando um componente estiver indisponível, de modo que sejam repassadas para ele quando se tornar disponível. Todavia, alguns sistemas não podem agir deste modo, pois precisam reagir imediatamente após a falha de algum componente. Em relação aos obstáculos na conectividade da rede, 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.

O *Extensible Messaging and Presence Protocol (XMPP)* [XMPP Standards Foundation 2007] é um exemplo de MOM, com tecnologia aberta,

que pode ser utilizado em aplicações de mensagens instantâneas, presença, *group chat*, video conferência e colaboração. Este protocolo evoluiu, se tornando um padrão para troca de mensagens assíncronas. Embora lide bem com os obstáculos na conectividade da Internet, o XMPP exige que o programador trabalhe com primitivas de baixo nível *send* e *receive* que não casam bem com a linguagem OO.

Com a finalidade de combinar objetos distribuídos com arquiteturas orientadas a eventos, minimizando os principais problemas de ambas as abordagens, Lima e colegas propuseram o *JIC – Java Internet Communication* [Lima et al. 2006]. O JIC é uma infraestrutura para comunicação de sistemas distribuídos que (i) é orientada a eventos; (ii) usa um modelo de comunicação assíncrono, não bloqueante, mas mantém uma semântica próxima do paradigma OO; (iii) cria um escopo bem definido para os fluxos de execução; (iv) lida bem com ambientes onde há presença de *firewalls* e redes privadas e (v) fornece um mecanismo de detecção de falhas simples.

O JIC foi utilizado em alguns projetos de código aberto, como por exemplo, a grade computacional *OurGrid* [Cirne et al. 2006] e o *grid information service NodeWiz* [Basu et al. 2009]. 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 [Lima 2006]. 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.

No entanto o JIC possui algumas lacunas, que impedem a adoção mais abrangente de sua tecnologia. Em particular, o JIC não possui uma semântica de falhas consistente, pois se houver perda de mensagens, a aplicação não será notificada da falha e poderá entrar em um estado inconsistente.

3. Formalização do Problema

Nesta seção, formalizaremos o problema da ausência de uma semântica de falhas consistente no JIC. Antes de abordar o problema propriamente dito, descreveremos a arquitetura geral e o funcionamento da detecção de falhas no JIC. Os principais elementos em um sistema desenvolvido sobre o JIC são o nó, o meio de comunicação e a conexão entre os nós.

Um **nó** representa um processo pertencente ao sistema distribuído. No *JIC*, um nó pode publicar diversos objetos remotos, que se comunicam com objetos de outros nós. Entretanto, abstrairemos a existência dos objetos na formalização deste problema, com fins de simplificar sua verificação formal. Desta forma, trataremos da comunicação nó – nó, em vez da comunicação objeto – objeto.

Dentre os componentes de um nó, destacamos a Aplicação e o Detector de Falhas. A Aplicação implementa a lógica de negócio de um nó e pode enviar mensagens de aplicação para os outros nós do sistema. A aplicação possui uma fila para o recebimento de mensagens e um fluxo de execução que consome estas mensagens.

O Detector de Falhas é um componente provido pelo *JIC*, que é ativado quando a aplicação registra interesse em um nó remoto. Este componente envia e verifica mensagens de controle, com o intuito de monitorar o estado da conexão com o nó remoto. O detector de falhas também possui uma fila de mensagens e um fluxo de execução.

Entre dois nós conectados, existe um **meio de comunicação**, que é o canal físico pelo qual trafegam mensagens de aplicação e de controle.

Quando a Aplicação de um nó pretende enviar mensagens para outro nó remoto, é preciso estabelecer uma **conexão** entre eles. As conexões possuem dois estados básicos. O estado UP indica que o nó remoto está presente ou disponível e a conexão com ele foi estabelecida. Por outro lado, o estado $DOWN$ indica a ausência do nó remoto. Existem outros estados intermediários que serão descritos posteriormente.

Dado um cenário onde existem dois nós A (emissor) e B (receptor), e A está enviando mensagens de aplicação para B, uma conexão pode ser vista como uma estrutura de dados distribuída, composta por:

- Um canal de dados por onde trafegam as mensagens de aplicação de A para B;
- Um canal de controle por onde trafegam as mensagens do detector de falhas de A para B;
- Um canal de controle por onde trafegam as respostas às mensagens do detector de falhas de B para A;
- Um *stub* em A, contendo a interface e o endereço de B e o estado da conexão. Este *stub* é um objeto que converte as invocações de métodos em mensagens para B;
- Um valor para a propriedade *heartbeat delay* em A, que define a periodicidade de envio de mensagens de controle de A para B;
- Um valor para a propriedade *detection time* em A, que define o tempo que detector de falhas de A espera pelas respostas de B antes de suspeitar da sua falha.

Devido à direção do canal de dados, convecionamos que esta conexão possui o sentido de A para B.

Simultaneamente, pode haver uma conexão no sentido de B para A. Assim sendo, existirá uma cópia da estrutura de dados no sentido inverso. Todavia, estas conexões são independentes e podem possuir valores de *heartbeat delay* e *detection time* diferentes. Além disto, o estado das conexões não é obrigatoriamente igual.

O estabelecimento normal de uma conexão de A para B no JIC ocorre através dos seguintes passos:

- A aplicação em A informa ao JIC que pretende se comunicar com B, registrando o interesse em B. Neste instante, é criado um *stub* de B em A, que ainda não pode ser utilizado, pois a conexão está marcada como $DOWN$;
- O detector de falhas de A começa a enviar periodicamente mensagens pelo canal de controle para B, segundo a propriedade *heartbeat delay*;
- Quando B se tornar disponível, responderá às mensagens de A pelo canal de controle;
- Após receber a resposta de B, o detector de falhas tornará o *stub* de B válido, marcando a conexão como UP;
- O detector de falhas informará à aplicação de A que B está disponível e lhe entregará um *stub* válido;
- A aplicação de A utilizará o *stub* de B para enviar mensagens pelo canal de dados;
- Por fim, quando B receber as mensagens do canal de dados, executará as ações definidas pela sua aplicação.

Após o estabelecimento de uma conexão, o JIC continua enviando mensagens de controle, de A para B, com o intuito de monitorar a disponibilidade de B. Se as respostas de B demorarem mais do que o tempo estipulado na propriedade *detection time* de A, o detector de falhas de A suspeitará da falha de B e informará à aplicação de A.

A aplicação poderá reagir após a detecção de uma falha conforme sua lógica de negócio definir. Como exemplo de reações que a aplicação pode efetuar diante de uma falha de parada, podemos citar:

- Abortar a computação que estava sendo feita e notificar o erro ao usuário;
- Substituir o nó falho por outro disponível;
- Esperar que o nó monitorado se recupere, para continuar o processamento.

Diante do cenário descrito, podemos concluir que a semântica de falhas do JIC está limitada à detecção de falhas por parada (*crash*) de um nó remoto. Esta semântica é inconsistente, pois não detecta a perda de mensagens de aplicação. Logo, um nó pode receber mensagens em uma ordem inesperada e entrar em um estado inconsistente, comprometendo o comportamento do sistema como um todo.

4. Protocolo de Conexão

Este artigo apresenta o *Commune*, uma plataforma de comunicação criada para resolver as lacunas do JIC. Neste trabalho em particular, detalharemos o Protocolo de Conexão do *Commune*, que possui uma semântica de falhas consistente, capaz de detectar a perda de mensagens de aplicação no meio de comunicação.

A fim de guiar a implementação do Protocolo de Conexão do *Commune*, definimos os seguintes requisitos de uma semântica de falhas consistente:

Dados dois nós *Commune*, A e B, se A está interessado em B, então:

Req 1: Se A e B estão operacionais, em algum momento será estabelecida uma conexão de A para B, que será mantida enquanto A e B estiverem operacionais e nenhuma mensagem enviada pelo canal de dados seja perdida.

Uma vez estabelecida a conexão entre A e B, então:

Req 2: Se B falhar, em algum momento A será notificado;

Req 3: Se uma mensagem enviada pelo canal de dados for perdida, em algum momento A será notificado da falha de B e, se também existir uma conexão de B para A, em algum momento B será notificado da falha de A.

O *Commune* possui uma estrutura interna semelhante à do JIC, conforme descrito na seção anterior. No entanto, adiciona o conceito de sessão às conexões entre os nós. Uma sessão é criada quando um nó se interessa por outro. Desta forma, as mensagens de controle que são enviadas para o nó remoto são marcadas com um número de sessão. Este mesmo número será utilizado pelas mensagens de aplicação no canal de dados.

Além do número de sessão, que é gerado aleatoriamente na criação da sessão, as mensagens também serão marcadas com um número de sequência, que inicia com o valor igual a zero. O número de sequência é incrementado no nó emissor, durante o envio de cada mensagem de aplicação.

O nó receptor mantém os números de sessão e sequência correntes. Portanto, este nó pode detectar se foi perdida alguma mensagem de aplicação quando o número de

sequência é maior que o esperado. 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 que o emissor da mensagem foi reiniciado, portanto se o receptor também está interessado no emissor, então a aplicação no receptor será notificada da falha do emissor.

Se o nó receptor das mensagens notar alguma perda de mensagem, a sessão será invalidada e as suas mensagens serão ignoradas. Desta forma, o emissor não receberá mais as respostas às mensagens de controle e ocorrerá um *timeout* da conexão.

Repetindo-se o cenário da seção 3, onde havia os nós A e B, se as mensagens de aplicação estão trafegando no sentido de A para B e de B para A, então existem duas conexões independentes. Os números de sessão e sequência das duas conexões não são obrigatoriamente iguais. A única dependência entre estas conexões ocorre durante a suspeita de uma falha. Se a falha for detectada pela conexão de A para B, a conexão de B para A também será invalidada. Por exemplo, se A detecta, como receptor, a perda de uma mensagem oriunda de B, então a conexão onde A é emissor também será invalidada.

Para verificar este protocolo de conexão, definimos, em termos formais, que uma **conexão em um nó** é uma tupla $C = \{L, R, s, r\}$, onde

L é o endereço do nó local,
R é o endereço do nó remoto,
s é o estado da conexão como emissor, 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;
- **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 desta disponibilidade. Neste caso, existe um número de sessão ativo e o número sequência é zero;
- **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 como receptor, 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.

Inicialmente, quando dois nós não estão se comunicando, não existem dados sobre a conexão nos nós. Modelamos este estado como $C_1 = \{L, R, Empty, Empty\}$. Vinte eventos que ocorrem no *Commune* podem interferir no estado de uma conexão entre dois nós. Estes eventos estão descritos na tabela 1.

A conexão pode ser iniciada pelo nó local (A) ou pelo nó remoto (B). Se o nó local decide iniciar a conexão, ele deve invocar o comando *registro de interesse*. O seu estado muda para $C_4 = \{A, B, Down, Empty\}$ e o seu detector de falhas começa a

enviar mensagens de controle – *está vivo?* – para o detector do nó remoto. Quando a primeira mensagem de controle chegar no nó remoto, o seu estado mudará para $C_2 = \{B, A, Empty, R(= 0)\}$ e enviará uma resposta à mensagem de controle – *atualizar estado* – para o nó local.

Após receber a resposta da mensagem de controle, o detector de falhas do nó local marca a conexão como $Uping$ e envia uma mensagem assíncrona de *notificação de recuperação* para a aplicação. A conexão não é marcada como Up diretamente porque o fluxo de execução do detector de falhas é diferente do fluxo da aplicação. Então, se o detector de falhas alterasse o estado das conexões diretamente, seria necessário que a aplicação sincronizasse todo o código fonte que utiliza *stubs*. Assim sendo, no momento que a aplicação consome a mensagem de notificação de recuperação, o próprio fluxo de execução da aplicação altera o estado da conexão para Up.

O detector de falhas pode suspeitar da falha de uma conexão que estava operacional quando a resposta das mensagens de controle tem o *tempo excedido* ou quando detecta uma perda de mensagem. Pelo mesmo motivo da notificação de recuperação, quando o detector de falhas suspeita de uma falha, ele não pode alterar o estado de uma conexão diretamente para $Down$. Ele marca a conexão como $Downing$ e envia uma mensagem assíncrona de *notificação de falha* para a aplicação. No momento que a aplicação consome a mensagem de notificação de falha, o próprio fluxo de execução da aplicação altera o estado da conexão para $Down$.

Quando uma conexão está marcada como Up, a aplicação pode utilizar o *stub* para *enviar mensagens* de aplicação pelo canal de dados. Estas mensagens serão recebidas e processadas pelo nó remoto. Definimos dois tipos de mensagens: as *mensagens sem callback* e as *mensagens com callbacks*. Um *callback* é um *stub* que é utilizado como parâmetro de uma mensagem no *Commune*. Os *callbacks* são bastante utilizados no *Commune* por causa do assincronismo, que não permite que os métodos dos *stubs* possuam retorno nem lancem exceção. Portanto, para receber as respostas de uma invocação remota, o nó que originou a comunicação precisa enviar uma referência de si mesmo para o nó remoto. Esta referência é um *callback*. Em termos de implementação, quando uma mensagem possui um *callback*, o receptor da mensagem recebe aquele *stub* com uma conexão já marcada como Up. Ou seja, o nó remoto já poderá utilizar o *stub* diretamente, sem precisar esperar pelo detector de falhas.

Quando uma aplicação não precisar mais se comunicar com um nó remoto, ela pode *liberar* um *stub*. Neste instante, todos os dados da conexão são removidos e o detector de falhas pára de enviar mensagens de controle para o nó remoto.

Levando em consideração todos estes eventos do *Commune*, modelamos uma máquina de estados com todos os estados e transições possíveis de um nó *Commune* isolado. Foram identificados 13 estados possíveis, oriundos do produto cartesiano $s\ x\ r$, menos os dois estados que se mostraram inatingíveis, $\{L, R, Downing, R(= 0)\}$ e $\{L, R, Downing, R(> 0)\}$. A figura 1 mostra os estados e transições da máquina de estados de um nó *Commune*.

Diversas sequências de transição podem ocorrer até que se chegue no nó que representa o estado $C_{13} = \{L, R, Up, R(> 0)\}$, no qual os dois nós estão trocando mensagens de aplicação e de controle.

Tabela 1. Eventos relativos ao Protocolo de Conexão do Commune

Mensagem	Parâmetros	Descrição
Registrar interesse	Endereço do nó remoto	A aplicação está se interessando em um nó remoto
Liberar	Stub	A aplicação está se desinteressando em um nó remoto
Está vivo?	Sessão ok, Sequência = 0	Recebida uma mensagem de controle de sessão não iniciada
	Sessão ok, Sequência ok	Recebida uma mensagem de controle de sessão com número de sequência válido
	Sessão ok, Sequência errada	Recebida uma mensagem de controle de sessão com número de sequência inválido, indica perda de mensagem
	Sessão diferente, Sequência = 0	Recebida uma mensagem de controle de uma outra sessão não iniciada
	Sessão diferente, *	Recebida uma mensagem de controle de uma outra sessão já iniciada, indica que ocorreu uma falha não notada
Atualizar estado	Sessão ok, nó UP	Recebido retorno de mensagem de controle, o nó remoto está disponível
	Sessão ok, nó DOWN	Recebido retorno de mensagem de controle, o nó remoto está indisponível
	Sessão errada, *	Recebido retorno de mensagem de controle, referente a outra sessão, indica que o nó remoto ainda não notou uma falha
Tempo excedido	Stub	O nó remoto, que estava disponível, parou de responder às mensagens de controle, suspeita de falha
Mensagem	Sessão ok, Sequência++	Recebida uma mensagem de aplicação com número de sequência incrementado (válido)
	Sessão ok, Sequência errada	Recebida uma mensagem de aplicação com número de sequência inválido, indica perda de mensagem
	Sessão errada, *	Recebida uma mensagem de aplicação com uma outra sessão já iniciada, indica que ocorreu uma falha não notada
Mensagem com Callback	Sessão ok, Sequência++	Recebida uma mensagem de aplicação com callback com número de sequência incrementado (válido)
	Sessão ok, Sequência errada	Recebida uma mensagem de aplicação com callback com número de sequência inválido, indica perda de mensagem
	Sessão errada, *	Recebida uma mensagem de aplicação com callback com uma outra sessão já iniciada, indica que ocorreu uma falha não notada
Notificar recuperação	Stub	A aplicação foi notificada da recuperação de um nó remoto
Notificar falha	Stub	A aplicação foi notificada da falha de um nó remoto
Enviar mensagem	Sequência++	O stub do nó remoto está sendo utilizado para mandar uma mensagem de aplicação com número de sequência incrementado

Para testar o Protocolo de Conexão do *Commune*, decidimos utilizar uma ferramenta de verificação formal. Escolhemos a ferramenta *Spin* [Holzmann 2003], por dois motivos principais. Em primeiro lugar, para definir os seus modelos, o *Spin* utiliza a linguagem *Promela* [Iosif 1998], 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.

Para se obter a prova dos requisitos, todo o cenário em estudo foi modelado na linguagem Promela. O modelo possui dois nós e quatro canais. Cada nó implementava o comportamento definido na máquina de estados da figura 1. Os canais possuem *buffer* com tamanho igual a um e são utilizados para o envio e recebimento de mensagens nas duas conexões.

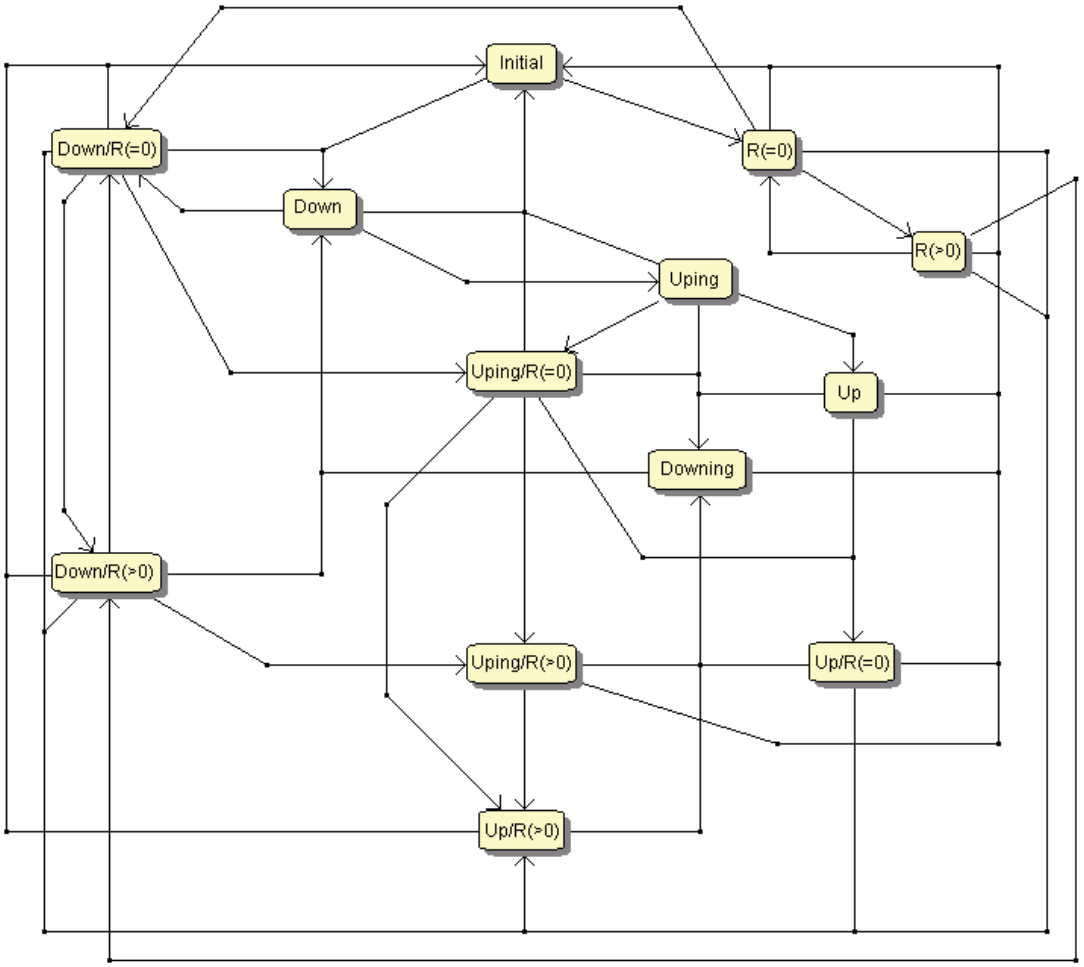

Figura 1. Máquina de estados de um nó Commune

Executamos a verificação do modelo a partir do estado inicial, com conexões e canais vazios. O *Spin* atingiu todos os estados do modelos e não encontrou *deadlocks*, o que significa que não existe um estado onde o modelo permanece infinitamente parado. Deste modo, verificamos que o Protocolo de Conexão não possui *deadlocks*.

A prova da ausência de *livelocks* no Protocolo de Conexão é mais complexa, pois significa que, a partir de qualquer estado atingível, é possível restabelecer a conexão, ou seja, atingir o estado $C_{13} = \{L, R, Up, R(> 0)\}$ em A e B. A linguagem *Promela* não possui semântica para verificar este tipo de assertiva numa única execução, portanto decidimos utilizar *scripts shell* para montar todos os cenários possíveis, verificando cada um individualmente.

Diferentemente da verificação de ausência de *deadlocks*, onde era preciso apenas verificar o modelo a partir do estado inicial, na verificação de ausência de *livelocks*, executamos 456.976 verificações. Este número é a multiplicação de 13 estados possíveis em A, 13 estados possíveis em B, 13 tipos de mensagens possíveis na conexão saindo de A, 4 tipos de mensagens possíveis na conexão chegando a A, 13 tipos de mensagens possíveis na conexão saindo de B e 4 tipos de mensagens possíveis na conexão chegando em B.

Foram executadas verificações iniciando a partir dos 456.976 estados possíveis. Em cada uma das verificações, não houve *deadlocks* e todos os estados foram atingidos. Portanto, concluímos que o estado $C_{13} = \{L, R, Up, R(> 0)\}$ em A e B simultaneamente sempre é atingível e não existem *livelocks*.

5. Projeto do Protocolo de Conexão

Neste artigo, foram detalhados os aspectos do *Commune* para solucionar a ausência de uma semântica de falhas consistente no *JIC*. Esta seção mostra a localização do protocolo de conexão na arquitetura do *Commune*, que está diagramada na Figura 2.

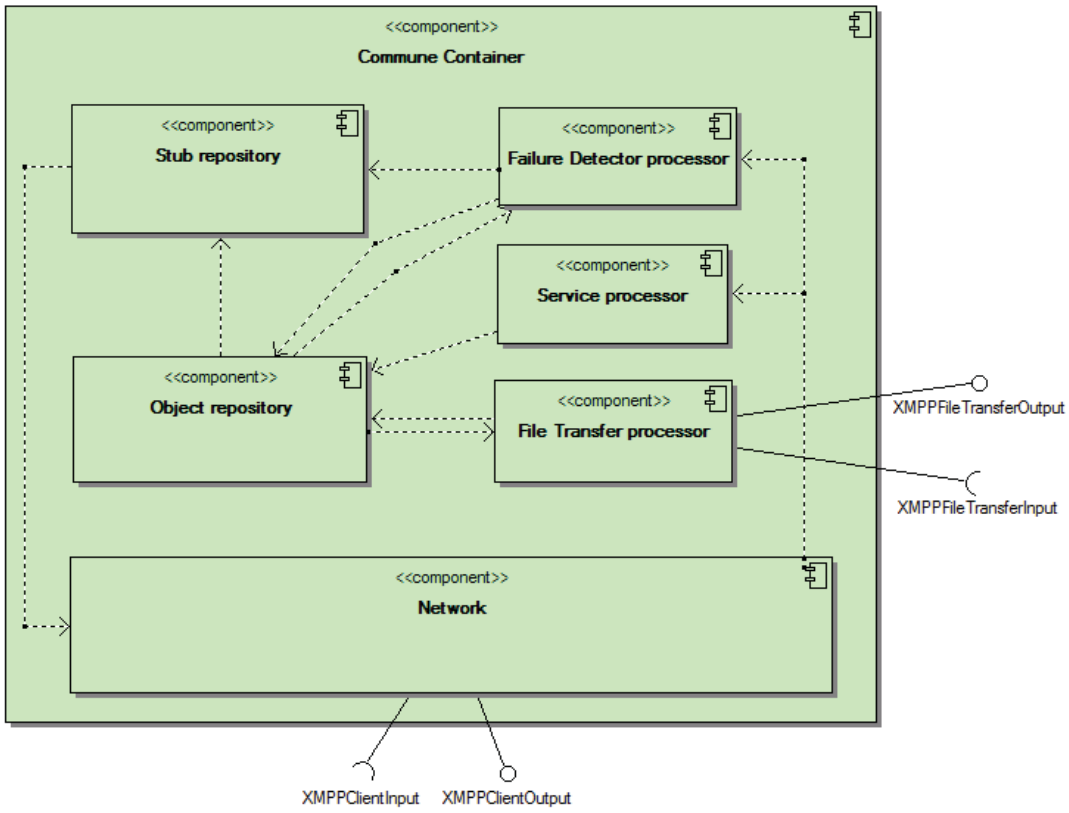

Figura 2. Arquitetura geral do Commune

Em um sistema distribuído feito sobre o *Commune*, cada nó é representado por um *Container*. O *Container Commune* é composto por seis componentes principais:

Object repository. Armazena os objetos que foram publicados no *Container*. Estes objetos receberão as mensagens remotas enviadas para o nó e implementarão a lógica da aplicação.

Stub repository. O componente *Stub repository* é responsável pelo armazenamento de todos os *stubs* ativos. Quando um *stub* tem alguns dos seus métodos invocados, automaticamente é gerada uma mensagem de saída para o componente *Network*.

Network. Este componente contém uma pilha de protocolos que são implementados pelo *Commune*. Cada camada do componente *Network* lida com um aspecto da

comunicação remota; define, portanto, um protocolo entre os nós. Por causa do *projeto* em camadas, a implementação destes protocolos se torna menos acoplada e mais simples. Além disto, a adição de novas camadas se torna trivial.

Na camada mais baixa deste componente, está localizada a conexão com o servidor XMPP, portanto ele é responsável pelo envio e recebimento de mensagens XMPP. Ao receber uma mensagem XMPP, o componente *Network* a transforma em uma mensagem *Commune* que ascende por todas as suas camadas, sendo validada por cada protocolo. Por fim, a mensagem é roteada para o *Service processor* ou para o *Failure Detector processor*. Quando um *stub* manda uma mensagem de saída para o componente *Network*, a mensagem deve atravessar todas as camadas até ser enviada pela conexão XMPP.

O Protocolo de Conexão que foi validado neste trabalho será implementado como mais uma camada do componente *Network*. Esta camada possuirá um repositório onde serão armazenadas as conexões, como também será responsável por definir os números de sessão e sequência nas mensagens enviadas e verificá-los nas mensagens recebidas. Se houver perda de mensagens, o Protocolo de conexão ativará a notificação de falhas do *Failure Detector processor*.

Service processor. Os três últimos componentes são os processadores, que lêem as mensagens oriundas do componente *Network* e atuam sobre os repositórios de objetos e *stubs*. O *Service processor* possui uma fila bloqueante de mensagens de aplicação que é povoada pelo componente *Network*. Em um laço infinito, um fluxo de execução lê as mensagens de aplicação e as transforma em invocações de métodos nos objetos publicados.

Failure Detector processor. É o componente responsável pela monitoração dos objetos remotos. O *Failure detector processor* utiliza o componente *Network* para enviar mensagens de controle, que implementam seu algoritmo de detecção de falhas. Em sentido contrário, o componente *Network* enfileira as mensagens de controle recebidas, que são consumidas pelo fluxo de execução do *Failure Detector processor*.

File Transfer processor. Por fim, este processador possui um acesso direto à conexão XMPP para utilizar a API de transferências de arquivos do XMPP. Assim sendo, este componente não precisa interagir com as camadas do componente *Network*.

6. Conclusão e Trabalhos futuros

Neste artigo, identificamos uma classe de sistemas distribuídos que possuem código bastante complexo e cujos nós precisam manter diversas conexões abertas simultaneamente. Estes sistemas não deveriam ser desenvolvidos com tecnologias que utilizem a abordagem síncrona para a comunicação remota, pois haveria estouro de memória. As plataformas assíncronas também não são ideais para estes sistemas pois não são bem integradas às linguagens OO e dificultam a modularização do código.

Neste trabalho, foi apresentado o *Commune*, uma plataforma para o desenvolvimento de sistemas distribuídos assíncronos que provê uma API bem integrada às linguagens OO. O *Commune* foi desenvolvido a partir de uma tecnologia semelhante, chamada *JIC*, que possuia algumas lacunas.

A principal contribuição deste trabalho foi resolver a lacuna da ausência de uma semântica de falhas consistente no *JIC*. Para o *Commune*, projetamos um Protocolo de

Conexão, que foi verificado formalmente com o uso da ferramenta *Spin*. Além disto, definimos os componentes da arquitetura do *Commune* que devem ser alterados para acomodar o Protocolo de Conexão.

Como trabalhos futuros, pretendemos implementar o Protocolo de Conexão no *Commune* e experimentá-lo em produção nos sistemas que utilizam o *JIC*, como o *OurGrid* e o *NodeWiz*.

Referências

Basu, S., Costa, L., Brasileiro, F., Banerjee, S., Sharma, P., and Lee, S.-J. (2009). Nodewiz: Fault-tolerant grid information service. *Peer-to-Peer Networking and Applications*.

Brasileiro, F., Greve, F., Tronel, F., Hurfin, M., and Narzul, J.-P. L. (2001). Eva: An event-based framework for developing specialized communication protocols. In *NCA '01: Proceedings of the IEEE International Symposium on Network Computing and Applications (NCA'01)*, page 108, Washington, DC, USA. IEEE Computer Society.

Cirne, W., Brasileiro, F., Andrade, N., Costa, L., Andrade, A., Novaes, R., and Mowbray, M. (2006). Labs of the world, unite!!! *Journal of Grid Computing*, 4(3):225–246.

Coulouris, G. and Dollimore, J. (2005). *Distributed systems: concepts and design*. Addison-Wesley Longman Publishing Co., Boston, MA, USA.

Holzmann, G. J. (2003). *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional.

Iosif, R. (1998). The promela language, http://www.dai-arc.polito.it/dai-arc/manual/tools/jcat/main/node168.html.

Lima, A. (2006). Combinando objetos distribuídos e arquiteturas orientadas a eventos em uma infra-estrutura de comunicação para sistemas distribuídos. Master's thesis, UFCG.

Lima, A., Cirne, W., Brasileiro, F., and Fireman, D. (2006). A case for event-driven distributed objects. In *8th International Symposium on Distributed Objects and Applications (DOA)*, pages 1705–1721, Berlin / Heidelberg. Springer.

McLean, S., Naftel, J., and Williams, K. (2002). *Microsoft .NET Remoting*. Microsoft Press.

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

Schmidt, D. C., Rohnert, H., Stal, M., and Schultz, D. (2000). *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley & Sons.

Sun Microsystems (2009). Threading, http://java.sun.com/docs/hotspot/threads/threads.html, acessado em 16/06/2009.

Wollrath, A., Riggs, R., and Waldo, J. (1996). A distributed object model for the java system. In *Conference on Object-Oriented Technologies*, pages 219–232, Toronto, Canada.

XMPP Standards Foundation (2007). Extensible messaging and presence protocol (XMPP), http://www.xmpp.org.