UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

LUIZ ANTONIO RODRIGUES

Extensão do Suporte para Simulação de Defeitos em Algoritmos Distribuídos Utilizando o Neko

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Profa. Dra. Ingrid Jansch-Pôrto

Orientadora

Porto Alegre, dezembro de 2006

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Rodrigues, Luiz Antonio

Extensão do Suporte para Simulação de Defeitos em Algoritmos distribuídos utilizando o Neko / Luiz Antonio Rodrigues – Porto Alegre: Programa de Pós-Graduação em Computação, 2006.

[100]p. + 1 CD-ROM.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2006. Orientadora: Ingrid Jansch-Pôrto.

1. Tolerância a Falhas. 2. Sistemas Distribuídos. 3. Neko. 4. Simulação. I. Jansch-Pôrto, Ingrid. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. José Carlos Ferraz Hennemann
Vice-Reitor: Prof. Pedro Cezar Dutra Fonseca
Pró-Reitora de Pós-Graduação: Profa. Valquiria Linck Bassani
Diretor do Instituto de Informática: Prof. Flávio Rech Wagner
Coordenador do PPGC: Prof. Carlos Alberto Heuser
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

AGRADECIMENTOS

Agradeço em primeiro lugar a minha família, pelo apoio moral e financeiro. Aos meus pais e a minha irmã, muito obrigado. Agradeço também a minha querida amiga e tia Rosa Maria pela ajuda didática e correções sugeridas neste texto. A todos os demais familiares que, de uma maneira ou de outra, me incentivaram a concluir este trabalho.

Quero agradecer a Universidade Federal do Rio Grande do Sul, na pessoa da professora Ingrid Jansch-Pôrto, minha orientadora, pela oportunidade de ingresso no programa de pós-graduação do Instituto de Informática da UFRGS. Agradeço de modo geral a todos os professores com os quais tive oportunidade de adquirir novos conhecimentos.

Agradeço a CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) e ao Instituto de Informática, pela concessão da bolsa de mestrado, sem a qual não seria possível realizar esta etapa.

Gostaria também de agradecer ao Peter Urbán, um dos criadores do Neko, que mesmo estando distante sempre respondeu as minhas dúvidas com rapidez e clareza. Thank you!

Não posso deixar de agradecer aos muitos amigos conquistados na universidade e na cidade. Aos amigos da República Paraná Jeysonn, Luciano e Guilherme, companheiros fies e parceiros de estudo, churrasco, festas e futebol. Aos companheiros do laboratório, em especial ao Tórgan pela atenção com que ouvia minhas dúvidas e lamentações.

Enfim, muito obrigado a todos que contribuíram em todos os inúmeros sentidos para conclusão deste projeto e para a conquista deste título.

"Escrevo sem pensar, tudo o que o meu inconsciente grita.
Penso depois: não só para corrigir, mas para justificar o que escrevi."

(Mário de Andrade)

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ACK	*ACKnowledgment*
EDF	*Ethernet Distribution Frame*
FDDI	*Fiber Distributed Data Interface*
FIONA	*Fault Injector Oriented to Network Applications*
IP	*Internet Protocol*
LS	*Large Scale*
SD	Sistema Distribuído
SSFNet	*Scalable Simulation Framework Network*
TCP	*Transmission Control Protocol*
UDP	*User Datagram Protocol*

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

O estudo e desenvolvimento de sistemas distribuídos é uma tarefa que demanda grande esforço e recursos. Por este motivo, a pesquisa em sistemas deste tipo pode ser auxiliada com o uso de simuladores, bem como por meio da emulação. A vantagem de se usar simuladores é que eles permitem obter resultados bastante satisfatórios sem causar impactos indesejados no mundo real e, conseqüentemente, evitando desperdícios de recursos. Além disto, testes em larga escala podem ser controlados e reproduzidos. Neste sentido, vem sendo desenvolvido desde 2000 um *framework* para simulação de algoritmos distribuídos denominado Neko. Por meio deste *framework*, algoritmos podem ser simulados em uma única máquina ou executados em uma rede real utilizando-se o mesmo código nos dois casos.

Entretanto, através de um estudo realizado sobre os modelos de defeitos mais utilizados na literatura, verificou-se que o Neko é ainda bastante restrito nesta área. A única classe de defeito abordada, lá referida como colapso, permite apenas o bloqueio temporário de mensagens do processo.

Assim, foram definidos mecanismos para a simulação das seguintes classes de defeitos: omissão de mensagens, colapso de processo, e alguns defeitos de rede tais como quebra de enlace, perda de mensagens e particionamento. A implementação foi feita em Java e as alterações necessárias no Neko estão documentadas no texto. Para dar suporte aos mecanismos de simulação de defeitos, foram feitas alterações no código fonte de algumas classes do *framework*, o que exige que a versão original seja alterada para utilizar as soluções. No entanto, qualquer aplicação desenvolvida anteriormente para a versão original poderá ser executada normalmente independente das modificações efetuadas. Para testar e validar as propostas e soluções desenvolvidas foram utilizados estudos de caso. Por fim, para facilitar o uso do Neko foi gerado um documento contendo informações sobre instalação, configuração e principais mecanismos disponíveis no simulador, incluindo o suporte a simulação de defeitos desenvolvido neste trabalho.

Palavras-Chave: tolerância a falhas, Neko, sistemas distribuídos, simulação

Extension to Support Failures in Distributed Algorithm Simulation Using Neko

ABSTRACT

The study and development of distributed systems is a task that demands great effort and resources. For this reason, the research in systems of this type can be assisted by the use of simulators, as well as by means of the emulation. The advantage of using simulators is that, in general, they allow to get acceptable results without causing harming impacts in the real world and, consequently, preventing wastefulness of resources. Moreover, tests on a large scale can be controlled and reproduced. In this way, since 2000, a framework for the simulation of distributed algorithms called Neko has been developed. By means of this framework, algorithms can be simulated in a single machine or executed in a real network, using the same code in both cases.

However, studying the most known and used failure models developed having in mind distributed systems, we realized that the support offered by Neko for failure simulation was too restrictive. The only developed failure class, originally named crash, allowed only a temporary blocking of process' messages.

Thus, mechanisms for the simulation of the following failure classes were defined in the present work: omission of messages, crash of processes, and some network failures such as link crash, message drop and partitioning. The implementation was developed in Java and the necessary modifications in Neko are registered in this text. To give support to the mechanisms for failure simulation, some changes were carried out in the source code of some classes of the framework, what means that the original version should be modified to use the proposed solutions. However, all legacy applications, developed for the original Neko version, keep whole compatibility and can be executed without being affected by the new changes. In this research, some case studies were used to test and validate the new failure classes. Finally, with the aim to facilitate the use of Neko, a document about the simulator, with information on how to install, to configure, the main available mechanisms and also on the developed support for failure simulation, was produced.

Keywords: fault tolerance, Neko, distributed systems, simulation

1 INTRODUÇÃO

A utilização de sistemas computacionais é uma realidade presente em praticamente todas as áreas do conhecimento. Com isso, é essencial que os sistemas sejam cada vez mais confiáveis, visto que seu mau funcionamento pode causar danos irreversíveis. São exemplos tradicionais da necessidade de mecanismos de tolerância a falhas os equipamentos médicos, de aviação, energia nuclear e exploração espacial, mas esta enorme difusão de sistemas automatizados trouxe características críticas também a sistemas de telefonia, comerciais e bancários. No final da década de 90, Laprie (1998) já estimava o custo da ocorrência de falhas em um sistema comercial na ordem de bilhões de dólares.

O projeto de sistemas tolerantes a falhas pode ser feito de diversas formas como, por exemplo, um sistema pode possuir um comportamento bem definido para o caso de ocorrer uma falha ou pode ainda mascarar a falha ao usuário (CRISTIAN, 1991). Em alguns casos, a ocorrência de falhas e a conseqüente indisponibilidade parcial do sistema são toleráveis, isto é, não ocasionam danos irreversíveis ou que possam colocar em risco os usuários. Entretanto, existe uma crescente necessidade por sistemas em que o funcionamento incorreto ou a indisponibilidade não são tolerados. Neste último caso, surge a necessidade de sistemas tolerantes a falhas, isto é, aqueles que continuam a realizar seu objetivo, mesmo que de forma reduzida, na presença de falhas, sejam elas de hardware ou software.

Com a evolução das tecnologias de rede, os sistemas têm apresentado uma grande tendência a tornarem-se distribuídos, sendo interligados por redes locais ou pela Internet. Os sistemas distribuídos (SDs) caracterizam-se por serem constituídos de componentes dispersos geograficamente e interconectados por uma rede de comunicação. A comunicação pode ser feita através de troca de mensagens pela rede (JALOTE, 1994). Entretanto, falhas de componentes ou dos canais de comunicação, se não forem devidamente tratadas, podem resultar na interrupção dos serviços providos.

Existem diversas vantagens na utilização de sistemas distribuídos como, por exemplo, o compartilhamento de informações e recursos, o aumento da velocidade de computação com a distribuição de tarefas, escalabilidade e confiabilidade. Além disso, pelo fato dos componentes do sistema estarem separados geograficamente, as falhas nestes componentes ocorrem independentemente umas das outras, evitando que o sistema venha a falhar como um todo. Entretanto, por se tratar de um número maior de componentes envolvidos, o número de falhas no sistema tende a aumentar consideravelmente.

O estudo e o desenvolvimento de sistemas distribuídos são tarefas que demandam grande esforço e recursos, recursos estes que nem sempre se encontram disponíveis nas fases iniciais de projeto. Além disso, é extremamente difícil testar e validar um SD em um ambiente real, devido à variedade de situações que são factíveis de acontecer. Assim, a pesquisa em sistemas distribuídos pode ser facilitada utilizando-se simuladores

e/ou emuladores. A vantagem de uso destas ferramentas é que elas permitem obter resultados bastante satisfatórios sem causar impactos indesejados no mundo real e, conseqüentemente, evitando desperdício de recursos. Além disto, podem ser realizados testes em larga escala que podem ser controlados e reproduzidos. Tem-se ainda um maior controle sobre o tempo, permitindo que fenômenos possam ser estudados de forma acelerada ou expandida. O uso de simuladores é, por este motivo, uma forma apropriada de se determinar, com resultados muito próximos da realidade, o comportamento que um SD pode apresentar e as conseqüências do mesmo. No entanto, o desenvolvimento de simuladores pode ser tão complexo quanto o próprio sistema em desenvolvimento. Por esta razão, há um crescente esforço da comunidade científica da área para criar ferramentas que sejam fáceis de manter e expandir, sem que haja grandes perdas de eficiência (CRAIG, 1996).

Visando facilitar o estudo de sistemas distribuídos, vem sendo desenvolvido desde 2000 um *framework* para simulação de algoritmos distribuídos denominado Neko (URBÁN, DÉFAGO e SCHIPER, 2001). Por meio deste *framework*, algoritmos podem ser simulados em uma única máquina ou executados em uma rede real, utilizando-se o mesmo código nos dois casos. Atualmente, Neko disponibiliza protocolos de redes reais, como TCP/IP, e simuladas. Para suporte ao estudo de defeitos, vem sendo desenvolvida uma biblioteca de algoritmos contendo detectores de defeito e mecanismos para simulação de defeitos. Entretanto, até o presente momento, apenas uma classe de defeitos foi abordada, e possibilita apenas o bloqueio temporário de mensagens. Na documentação do Neko, ela é denominada como defeito por colapso (*crash*), embora tal comportamento não se enquadre nas definições de Cristian (1991) ou Birman (1996), utilizadas como referência neste trabalho. Assim, estão em aberto as implementações de defeitos: de colapso por parada, pausa, amnésia e amnésia parcial (CRISTIAN, 1991); e de omissão no envio e no recebimento, temporização e rede (BIRMAN, 1996; CRISTIAN, 1991). Como estes autores utilizam o termo defeito (*failure*) em sua classificação ao invés de falha (*fault*) para representar o mau funcionamento do sistema, o restante deste trabalho seguirá a nomenclatura dos mesmos. No entanto, o termo falha ainda será utilizado num contexto mais geral, enquanto que defeito fará referência ao comportamento errado de um componente específico do sistema.

Por meio deste trabalho, pretende-se ampliar o modelo de simulação de defeitos oferecido pelo Neko e, assim, fornecer uma ferramenta mais completa para o estudo do impacto das falhas dentro do ambiente de pesquisa do Grupo de Tolerância a Falhas da UFRGS. As classes de defeitos adicionadas incluem as de colapso por parada e por pausa, omissão no envio, omissão no recebimento, rede e particionamento de rede.

1.1 Trabalhos Relacionados

O trabalho de Trindade (2003) investigou a possibilidade de utilizar o simulador de redes NS-2 (2005) em simulações de ambientes distribuídos envolvendo cenários com defeitos. Assim como o Neko, o NS-2 tem código aberto e pode ser estendido. Embora seja destinado ao estudo de redes de computadores, o trabalho conseguiu mostrar que o simulador pode ser útil na simulação de defeitos em um sistema distribuído.

Diferentemente do NS-2, que possui ênfase na rede, o Neko foi criado com o objetivo de permitir a simulação e experimentação de algoritmos desenvolvidos para sistemas distribuídos. Entretanto, por se tratar de um projeto recente, o mesmo vem sendo constantemente incrementado e alguns trabalhos vêm sendo publicados relatando estes melhoramentos.

Uma proposta apresentada por Urbán, Défago e Katayama (2003) visa desenvolver uma ferramenta de estudo em sistemas de larga escala (LS – *Large Scale*) a partir da estrutura já definida no Neko. O objetivo é reestruturar o *framework* para refletir as propriedades dos sistemas de larga escala, disponibilizando modelos de rede mais complexas, mecanismos de tolerância a falhas nos nodos utilizados na emulação e estudos sobre a escalabilidade da ferramenta.

Seguindo esta linha, Matsushita (2005) realizou a integração do Neko com o SSFNet (*Scalable Simulation Framework Network*) (SSFNET, 2005), permitindo a simulação de modelos de rede mais realísticos no Neko. Com esta integração, o Neko pode utilizar todos os protocolos de rede incluídos no SSFNet, incluindo a possibilidade de definição de topologias e endereçamento condizentes com os sistemas de larga escala. Para efetuar a integração, foram desenvolvidos mecanismos de interfaceamento entre as duas ferramentas, incluindo: a sincronização entre os escalonadores das duas ferramentas; mapeamento dos diferentes tipos de endereçamento; e conversão do tipo de mensagem transmitida por cada modelo. Esta integração já foi incluída na nova versão do Neko (versão 0.9), embora todos os mecanismos para simulação da versão anterior continuem disponíveis. A emulação é realizada da mesma forma nas duas versões.

No que diz respeito a modelos de defeitos, a dissertação de Jacques-Silva (2005) abordou a injeção de falhas em aplicações Java de pequena e larga escala. A ferramenta gerada, denominada FIONA (*Fault Injector Oriented to Network Applications*), permite a injeção de falhas de omissão, duplicação, colapso de enlace (*link*) e de nodo, temporização e particionamento. O trabalho também se baseou no modelo de defeitos de Birman (1996).

As soluções apresentadas no presente trabalho já vêm sendo utilizadas por Balbinot (2006), membro do Grupo de Tolerância a Falhas da UFRGS, para gerar defeitos de omissão e avaliar o comportamento de um serviço de detecção de defeitos não confiável.

1.2 Organização do Trabalho

Além do que foi apresentado neste capítulo introdutório, o texto desta dissertação tem a seguinte organização:

O capítulo 2 descreve os dois modelos de defeitos utilizados no trabalho. O primeiro modelo foi apresentado por Cristian (1991) e descreve os defeitos de omissão, temporização, resposta, colapso e arbitrários. O segundo modelo, descrito por Birman (1996) divide os defeitos em parada, *fail-stop*, omissão no envio, omissão no recebimento, rede, particionamento, temporização e bizantinos.

O capítulo 3 aborda a estrutura e funcionamento do *framework* Neko, incluindo o modelo de programação, estrutura de rede e suporte para simulação de defeitos existente.

O capítulo 4 apresenta a modelagem e propostas de solução para a inclusão de defeitos em simulações no Neko. Os defeitos abordados incluem colapso por parada e por pausa, omissão no envio e no recebimento, rede e particionamento. Além disso, são analisados os demais defeitos discutidos no capítulo 2, com sugestões para a simulação dos mesmos no Neko.

O capítulo 5 trata dos aspectos relacionados a implementação das soluções apresentadas para a simulação de defeitos, incluindo os ajustes necessários para refletir a modelagem proposta no capítulo 4.

No capítulo 6, é exemplificado o uso de todas as classes através de estudos de caso, demonstrando os resultados obtidos com a simulação de aplicações teste no ambiente de defeitos desenvolvido.

Por fim, são apresentadas as conclusões e as sugestões para trabalhos futuros.

No CD-ROM do anexo B estão disponíveis: a versão do Neko utilizada no desenvolvimento, os códigos-fonte escritos, os testes realizados, o manual e o texto desta dissertação, juntamente com os artigos submetidos/publicados.

2 MODELOS DE DEFEITOS

A tarefa de projetar e desenvolver um sistema distribuído que seja tolerante a falhas é extremamente difícil. Esta dificuldade é ocasionada pela quantidade de componentes envolvidos, que incluem, além daqueles presentes em um sistema centralizado, os demais componentes necessários para a comunicação entre os processos, incluindo a estrutura de rede (CRISTIAN, 1991).

Para determinar o quão tolerante a falhas será o sistema que se deseja desenvolver é preciso determinar que tipos de defeitos este sistema será projetado para tolerar, isto é, que modelo de defeitos será utilizado. Geralmente, o modelo de defeitos suportado pelo sistema terá influência direta na sua eficiência e complexidade. Duas classificações de defeitos são bastante aceitas atualmente: a de Cristian (1991) e a de Birman (1996), as quais são descritas a seguir. Cristian é tradicionalmente referenciado para defeitos em nodos e Birman apresenta aspectos bastante completos no âmbito dos defeitos de comunicação.

2.1 Modelo de Defeitos Descrito por Cristian

Cristian (1991) faz uso de três conceitos que ele considera fundamentais para o melhor entendimento da arquitetura dos sistemas distribuídos: serviço, servidor e a relação de dependência entre servidores. Um serviço especifica as operações que são executadas a partir de uma entrada ou em um tempo pré-determinado. Uma operação pode resultar em uma saída para um usuário e em uma mudança de estado do serviço. Um servidor é responsável pela execução de um ou mais serviços. A forma de execução de um serviço, bem como o seu estado, é omitida do usuário, que necessita saber apenas o seu comportamento. Servidores podem realizar seu serviço utilizando outros serviços fornecidos por outros servidores. Assim, a relação de dependência de um servidor *u* com um servidor *r* é estabelecida se o comportamento correto de *u* depende do comportamento correto de *r*.

Desta forma, para realizar sua classificação, Cristian define que “um servidor projetado para fornecer um serviço está correto se, em resposta às entradas, ele se comporta de maneira consistente com a especificação do serviço” e um defeito ocorre quando esta consistência não é mantida.

As seguintes classes de defeitos são definidas por Cristian:

- defeito de omissão (*omission failure*): ocorre quando um servidor omite resposta para uma entrada;
- defeito de temporização (*timing failure*): ocorre quando a resposta do servidor é funcionalmente correta, mas ocorre fora do intervalo de tempo especificado. Este defeito pode ser por antecipação da resposta (*early timing failure*) ou por atraso (*late timing failure*);

- defeito de resposta (*response failure*): ocorre quando o servidor responde incorretamente, podendo ocasionar um valor de saída incorreto (*value failure*) ou uma transição que leve o sistema a um estado incorreto (*state transition failure*);
- defeito de colapso (*crash failure*): ocorre quando o servidor pára de responder, podendo ou não ser reinicializado. Esta classe de defeitos está dividida em quatro subclasses:
  - colapso com amnésia (*amnesia-crash*): ocorre quando o servidor reinicializa em um estado inicial pré-definido, descartando as entradas anteriores ao colapso. Como exemplo, pode-se citar a reinicialização do sistema operacional;
  - colapso com amnésia parcial (*partial-amnesia-crash*): ocorre quando parte do estado anterior ao colapso é recuperado na reinicialização, enquanto que o restante é inicializado com valores pré-definidos;
  - colapso por pausa (*pause-crash*): ocorre quando o servidor é reinicializado no último estado anterior ao colapso;
  - colapso por parada (*halting-crash*): ocorre quando o servidor não pode ser reinicializado após o colapso.
- defeitos arbitrários: quando o sistema pode manifestar qualquer comportamento apresentado pelas classes definidas anteriormente, de forma aleatória.

2.2 Modelo de Defeitos Descrito por Birman

De acordo com Birman (1996), um sistema distribuído é um conjunto de programas executando em um ou mais componentes e coordenando ações através de troca de mensagens. Uma rede de computadores é uma coleção de computadores interconectados por hardware que permite a troca de mensagens. Esta troca é gerenciada por um protocolo. Assim como um programa é um conjunto de instruções e um processo descreve a execução destas instruções, um protocolo é responsável por coordenar a comunicação em um programa distribuído. Desta forma, um sistema distribuído é o resultado da execução de um conjunto de protocolos para coordenar as ações de uma coleção de processos na rede.

Baseado nestes conceitos, Birman (1996) estabelece as classes de defeitos para um sistema distribuído, definidas como segue:

- defeito por parada (*halting failure*): neste modelo, um processo ou computador pára sua execução sem realizar qualquer ação incorreta. Neste caso, a única maneira de se deduzir que o sistema parou é através de *timeouts*, seja pela ausência de mensagens "*keep alive*", seja pela ausência de respostas às mensagens de *ping*;
- defeito do tipo *fail-stop* (*fail-stop failure*): neste caso, o processo que apresenta defeito pára de forma semelhante ao modelo *halting failure*. Entretanto, os defeitos são relatados aos demais membros envolvidos com o componente que apresentou defeito por meio de um serviço de notificação;
- defeito de omissão no envio (*send omission failure*): ocorre quando uma mensagem deixa de ser enviada, mesmo que o processo emissor acredite que a mesma tenha sido. Estes defeitos são geralmente causados por falta de espaço no *buffer* do sistema operacional ou da interface de rede, causando o descarte após o envio ter sido feito pela aplicação e antes de ser realizado pelo computador emissor;

- defeito de omissão no recebimento (*receive omission failure*): é semelhante ao defeito por omissão no envio. Entretanto, nesta classe, a perda da mensagem ocorre próxima ao processo destino. Acontece freqüentemente devido a uma falha de memória do *buffer* ou devido à detecção de erro nos dados recebidos;
- defeito de rede (*network failure*): ocorre quando a rede perde mensagens trocadas entre pares de processos;
- defeito de particionamento de rede (*network partitioning failure*): ocorre quando a rede é subdividida em duas ou mais sub-redes. As mensagens trocadas entre os processos da mesma sub-rede continuam trafegando, mas as mensagens entre sub-redes diferentes não podem ser transmitidas até o restabelecimento da comunicação. É uma das classes de defeitos mais difíceis de tratar, já que os processos não conseguem distinguir se o processo com o qual estavam se comunicando entrou em colapso ou se ocorreu uma falha de comunicação;
- defeito de temporização (*timing failure*): ocorre quando uma propriedade temporal do sistema é violada. Por exemplo, quando uma ação é tomada de forma antecipada ou atrasada ou quando uma mensagem sofre um atraso maior que o máximo tolerado pela conexão de rede;
- defeito bizantino (*byzantine failure*): esta classe inclui uma variedade de outros comportamentos com defeito, incluindo corrupção de dados, programas que falham em seguir o protocolo adequado e qualquer comportamento malicioso ou adverso que force o sistema a violar suas propriedades de confiabilidade (*reliability*).

2.3 Comparação entre os Modelos

Os defeitos de colapso por parada definidos por Cristian são equivalentes aos defeitos de parada descritos por Birman. Entretanto, Cristian divide os defeitos de colapso em quatro categorias, dependendo do estado do componente após sua reinicialização. Isto permite identificar o comportamento interno do componente após o colapso.

Birman define ainda os defeitos de comunicação, incluindo defeitos de rede e de particionamento de rede, não abordados diretamente por Cristian. Para Cristian, a perda de mensagens ocasionadas pelo serviço de comunicação pode ser considerada como um defeito de omissão.

Por fim, é apresentada por Birman a classe de defeitos bizantinos, definida por Cristian como arbitrários. Esta classe abrange todas as outras classes, isto é, se um sistema é projetado para tratar defeitos bizantinos, é capaz de se recuperar de qualquer tipo de defeito. Entretanto, esta é uma propriedade muito difícil de ser alcançada em um sistema real, justamente pela sua abrangência e imprevisibilidade.

As classes de defeitos podem ainda estar relacionadas e organizadas de forma hierárquica. Na Figura 2.1, são ilustradas as classes de defeitos e a relação entre elas. Os defeitos de colapso são os mais simples e mais bem definidos. Segundo Jalote (1994), esses defeitos são englobados pelos defeitos de omissão, visto que um colapso por parada pode ser generalizado como uma omissão de tempo infinito. Já os defeitos de omissão são englobados pelos de temporização e estes pelos defeitos arbitrários. Isto significa que os mecanismos desenvolvidos para tratar defeitos em classes mais abrangentes, também podem ser utilizados para as classes que são englobadas. Como

exemplo, se um sistema é projetado para tratar defeitos bizantinos, é capaz de tratar qualquer outra classe de defeitos.

Figura 2.1: Classes de defeitos (JALOTE, 1994)

É importante salientar que Jalote (assim como Birman) considera apenas os defeitos de colapso por parada, no qual o sistema é suspenso permanentemente. Esta classificação não inclui as demais classes de colapso descritas por Cristian, já discutidas anteriormente. A classe de defeitos de resposta representa os defeitos ocasionados pela geração de respostas incorretas, mesmo a partir de entradas corretas, isto é, defeitos gerados pela corrupção do conteúdo das mensagens. Esta subclasse é englobada pelos defeitos bizantinos, mas não mantém relação com as demais classes, visto que não ocasionam impacto no tempo de resposta.

No restante deste trabalho serão abordadas as seguintes classes, já descritas acima:

- As quatro classes de colapso, descritas por Cristian;
- As classes de omissão no envio e no recebimento, rede e particionamento propostas por Birman;

O defeito de resposta não será abordado porque pode ser tratado eficientemente por um protocolo de baixo nível e, portanto, não causa impacto na aplicação final. O defeito de temporização, embora importante, não foi incluído no estudo devido às restrições de tempo para o desenvolvimento de modelos de temporização e testes da solução. O defeito arbitrário ou bizantino pode ser obtido com a utilização arbitrária e/ou conjunta dos demais defeitos abordados no trabalho, mas também não será incluído no estudo em função do tempo necessário para realizar testes significativos.

3 O FRAMEWORK NEKO

Assim como qualquer sistema computacional moderno, os sistemas distribuídos estão constantemente necessitando de maior desempenho e confiabilidade. Isto tem incentivado o desenvolvimento de técnicas e ferramentas que permitam estudar novos algoritmos e protocolos distribuídos mais eficientemente, isto é, em menor tempo e a custos reduzidos.

Segundo Urbán, Défago e Schiper (2001) existem três enfoques para avaliar o desempenho de algoritmos:

1. *enfoque analítico*: fornece resultados baseados em um modelo parametrizado do ambiente de execução;
2. *simulação*: os resultados são coletados através da execução do algoritmo em um ambiente simulado, utilizando-se geralmente de um modelo estocástico;
3. *medição*: os resultados são obtidos a partir da execução do algoritmo em um ambiente real.

A vantagem de se usar simuladores é que eles permitem obter resultados satisfatórios sem causar impactos indesejados no mundo real e, conseqüentemente, evitando desperdícios de recursos. Além disto, podem ser realizados testes em larga escala que podem ser controlados e reproduzidos. Por outro lado, a medição possibilita a captura de detalhes importantes e que poderiam acabar sendo omitidos em uma simulação. Assim, o ideal seria conciliar os dois modelos.

No entanto, a grande dificuldade reside no fato de que, muitas vezes, é necessário desenvolver diferentes implementações para medição e simulação, pois nem sempre estas duas abordagens podem ser realizadas utilizando a mesma forma básica de descrição. Visando diminuir estes problemas, Urbán, Défago e Schiper (2001) propuseram e iniciaram o desenvolvimento do *framework* Neko, o qual visa fornecer um ambiente que permita tanto a simulação de algoritmos distribuídos, quanto a execução dos mesmos em um ambiente real, empregando a mesma implementação, neste caso, escrita em Java.

3.1 Arquitetura do Neko

A arquitetura do *framework* é dividida basicamente em duas partes principais: aplicação e rede (Figura 3.1). No nível da aplicação, os processos comunicam-se utilizando troca de mensagens. O emissor envia os dados através da rede utilizando primitivas `send()`, assíncronas, e a rede disponibiliza estes dados ao receptor através da primitiva `deliver()`. Todos os processos são programados em multicamadas, o que facilita a adição de funcionalidades aos mesmos. Um exemplo de programação multicamadas fornecido por Neko é o paradigma coordenador-trabalhador, em que um processo coordenador divide uma tarefa em subtarefas e as distribui entre os

trabalhadores. Após processar a tarefa, cada trabalhador envia o resultado obtido ao coordenador. Neste exemplo, pode-se adicionar uma camada superior que implementa o coordenador e os trabalhadores e uma segunda camada inferior que implementa algum mecanismo de tolerância a falhas como um detector de defeitos. Esta segunda camada fica responsável por detectar e, possivelmente tratar defeitos nos trabalhadores. Caso um trabalhador pare de responder, o coordenador é avisado e realoca a tarefa para um outro trabalhador. A hierarquia de camadas facilita o entendimento e a modelagem do sistema proposto e fornece um mecanismo simples para possíveis alterações no seu comportamento.

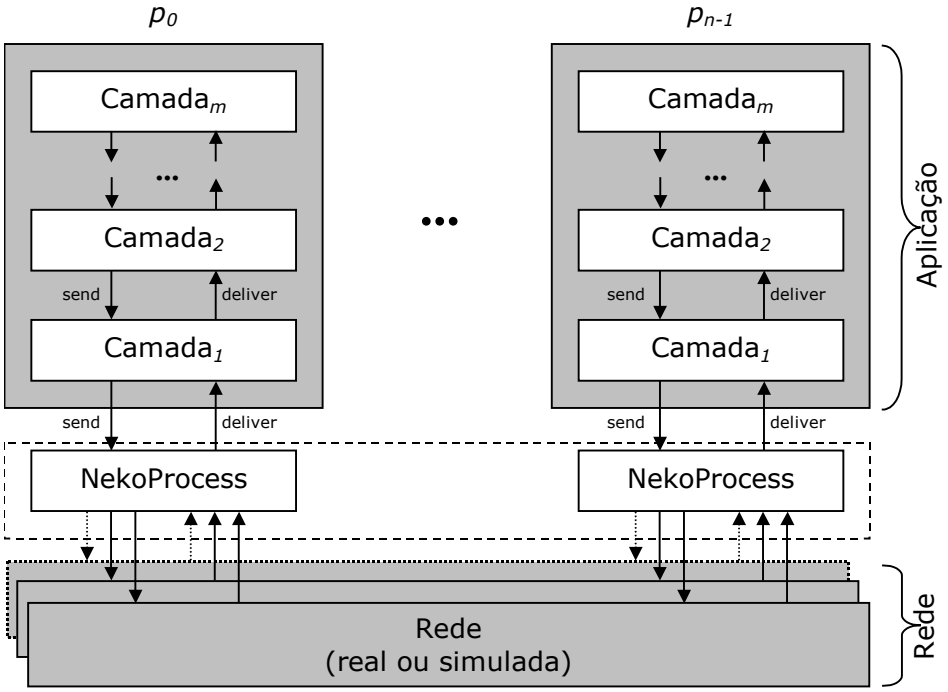


Figura 3.1: Arquitetura do *framework* Neko (URBÁN, DÉFAGO e SCHIPER, 2001)

O segundo componente da ferramenta é a rede. A infra-estrutura de comunicação pode ser controlada de diferentes maneiras, podendo ser instanciada a partir de uma coleção de redes previamente definidas (como por exemplo, uma rede TCP/IP real ou uma rede *Ethernet* simulada) ou ainda por meio da extensão e adição de novos modelos à ferramenta. Além disso, Neko pode gerenciar diversas redes em paralelo.

Como já citado anteriormente, as aplicações são construídas seguindo uma hierarquia de camadas. As mensagens são passadas através das camadas por meio do método `send()` e são entregues através do método `deliver()`. As camadas podem ser ativas ou passivas. Camadas passivas não possuem *threads* de controle próprias, o que exige que as camadas inferiores utilizem o método `deliver()` para receber os dados. As camadas ativas são derivadas da classe `NekoThread` e possuem suas próprias *threads* de controle, possibilitando que recebam as mensagens das camadas inferiores através do método `receive()`.

Cada processo na aplicação possui um objeto `NekoProcess` associado (Figura 3.2), estabelecido entre as camadas de aplicação e rede. Este objeto é responsável por enviar e receber mensagens da rede de comunicação, efetuar o registro destas mensagens no arquivo de *log*, criar a pilha de camadas do processo no início da simulação e manter o identificador do processo ao qual está vinculado. Em uma simulação, todos os processos estão localizados em uma única máquina. Em uma execução distribuída, cada processo pode estar localizado em uma máquina diferente.

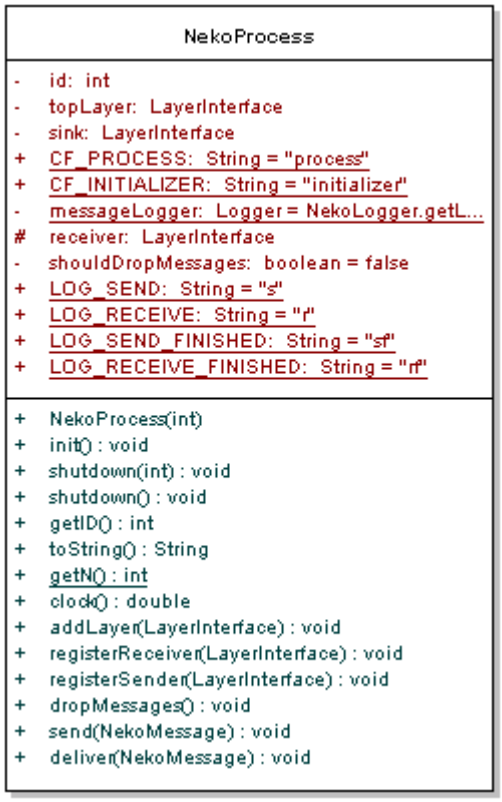


Figura 3.2: Diagrama da classe lse.neko.NekoProcess

As mensagens transmitidas através das primitivas `send()`, `deliver()` e `receive()` são instâncias da classe `NekoMessage` (Figura 3.3). O envio de mensagens pode ser do tipo *unicast* ou *multicast*. Cada mensagem possui os seguintes campos:

- *network*: identifica a rede que será utilizada para enviar a mensagem;
- *from*: identificador do processo que deu origem a mensagem;
- *to*: lista dos processos que receberão a mensagem;
- *type*: identificador do tipo da mensagem;
- *content*: conteúdo da mensagem, constituído por um objeto Java qualquer.

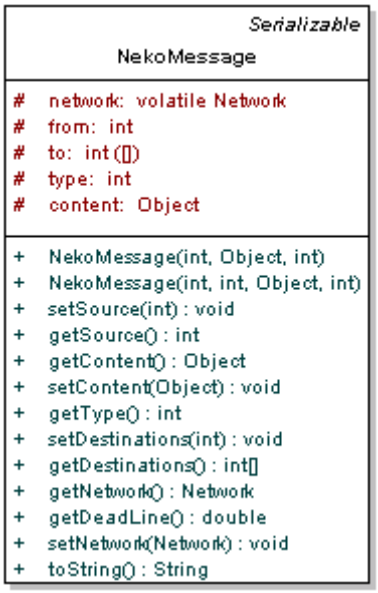


Figura 3.3: Diagrama da Classe `lse.neko.NekoMessage`

3.2 Estrutura de Rede

A rede no Neko constitui a camada de transporte da arquitetura (Figura 3.1). Como já mencionado anteriormente, Neko possibilita a utilização de redes reais e simuladas sem que haja necessidade de qualquer alteração no código fonte da aplicação.

Todas as redes, reais ou simuladas, estendem (direta ou indiretamente) a classe `lse.neko.Network`. (Figura 3.4). O desenvolvedor deve adicionar o comportamento da rede ao método `send()`, que é herdado da interface `LayerInterface` e será responsável pela entrega de mensagens aos processos participantes do experimento.

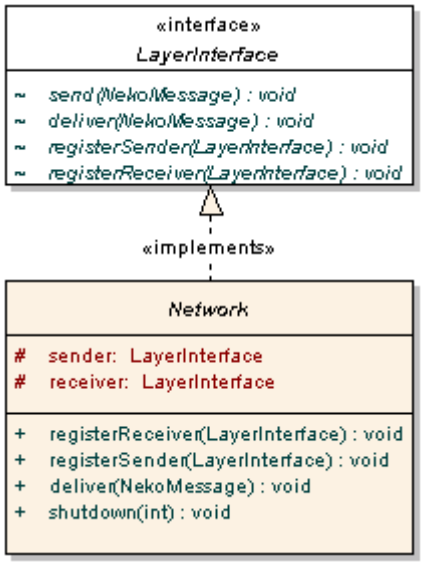


Figura 3.4: Diagrama da Classe `lse.neko.Network`

As redes reais disponibilizadas para emulação no Neko estão apresentadas na Figura 3.5. Todas elas são implementadas a partir da biblioteca de *Sockets* Java. Portanto, existem basicamente duas classes de rede emuladas: as que utilizam o protocolo TCP (*Transmission Control Protocol*) e as que utilizam o protocolo UDP (*User Datagram Protocol*). Com exceção da classe `lse.neko.networks.comm.UDPNetwork`, todas as demais redes utilizam o protocolo TCP. Por se tratar de máquinas reais, os parâmetros de IP (*Internet Protocol*) e número de porta de cada processo (máquina) participante são informados na inicialização do Neko, isto é, no início da execução do experimento.

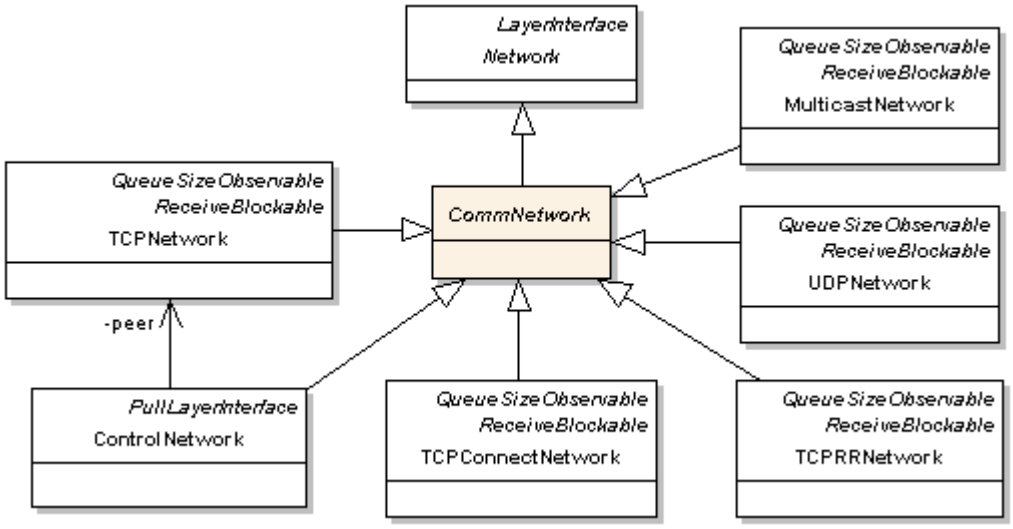


Figura 3.5: Diagrama das classes de rede reais disponíveis no Neko

As redes simuladas (Figura 3.6) incluem versões simplificadas de *Ethernet*, FDDI[1] e EDF (*Ethernet Distribution Frame*). Além disso, existem outros modelos que simulam redes mais simples, como a `BasicNetwork` que utiliza um parâmetro fixo de atraso para a transmissão de mensagens e a `RandomNetwork` que configura o atraso randomicamente.

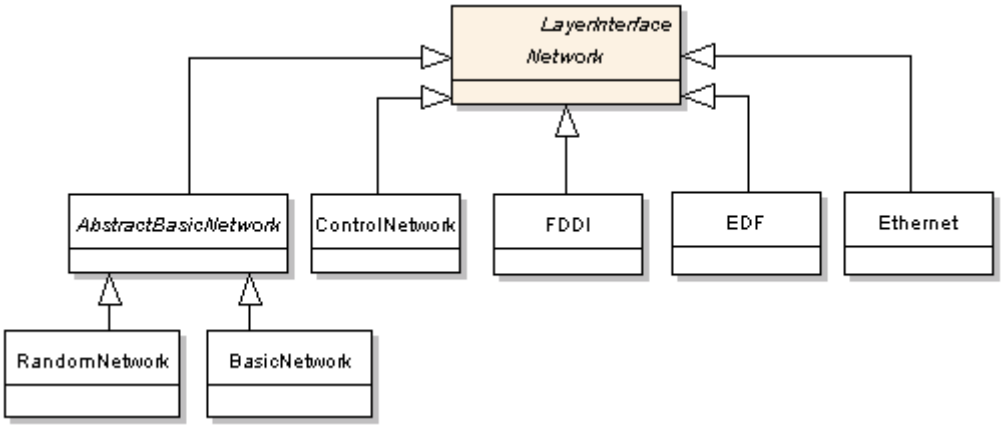


Figura 3.6: Diagrama das classes de rede disponíveis para simulação no Neko

A definição do modelo de rede a ser utilizado é feita através do arquivo de configuração por meio do parâmetro `network`. Por exemplo, para utilizar uma rede real do tipo TCP disponível na versão 0.8 do Neko basta adicionar o parâmetro `network = lse.neko.networks.comm.TCPNetwork` ao arquivo de configuração.

O suporte de rede no Neko, até a versão 0.8, não permite a definição de topologia. A camada de rede repassa as mensagens entre dois processos como se os mesmos estivessem conectados diretamente. Este cenário era bastante simples e não condizente com a realidade da maioria dos sistemas distribuídos, que incluem redes e sub-redes com roteadores, algoritmos de controle de congestionamento, entre outros. Obviamente que, quando se fala em topologia, estão sendo levadas em consideração apenas as redes simuladas, visto que as redes emuladas utilizam a própria topologia física da rede real.

[1] FDDI (*Fiber Distributed Data Interface*): é a especificação de uma LAN do tipo anel duplo com passagem de *token*. A arquitetura de anel duplo (anel primário e secundário) fornece maior confiabilidade e robustez a rede.

A afirmação de que não existe possibilidade de definição de topologia advém do fato de não existem mecanismos para criação de enlaces, roteadores, políticas de enfileiramento e demais elementos comumente utilizados nas redes reais. Até mesmo no modelo de rede FDDI, que embora represente uma rede em anel, não há o registro da mensagem sendo repassada em cada nodo do anel. Existe apenas o cálculo do tempo necessário para que a mensagem alcance o destino, levando em consideração o número de saltos (*hops*) entre eles.

Visando aproximar ainda mais a simulação da realidade das redes atuais, fez-se necessário acrescentar mecanismos para permitir uma configuração mais específica da topologia de rede. A solução apresentada por Matsushita (2005) e incluída na versão 0.9 do Neko propõe a utilização em conjunto do Neko e do simulador de redes SSFNet (SSFNET, 2005). Este simulador é escrito em Java e disponibiliza diversos modelos de protocolos (IP, TCP, UDP, BGP4, OSPF), elementos de rede (*hosts*, *routers*, *links*, LANs) e suporte para modelagem e simulação de arquiteturas de rede multi-protocolo e multi-domínio, como a Internet.

Desta forma, foram incluídos no Neko mecanismos que permitem sua comunicação com o SSFNet e a utilização de toda a infra-estrutura já desenvolvida no mesmo. Este uso conjunto não é obrigatório e os experimentos ainda podem ser realizados independentemente do SSFNet. As redes existentes na versão anterior ainda estão disponíveis e podem ser empregadas normalmente.

Para efetuar a ligação entre o Neko e o SSFNet foram criados modelos de redes específicos que fazem uso dos parâmetros necessários. Um destes parâmetros é o arquivo DML, utilizado para configurar o SSFNet e que inclui a definição da topologia. Foram também realizadas algumas adaptações para que a interação entre os simuladores fosse possível. As principais adaptações incluem sincronização do escalonador, mapeamento entre os diferentes esquemas de endereçamento e conversão do tipo da mensagem transmitida em cada simulador.

3.3 Suporte a Simulação de Defeitos

Devido a complexidade de desenvolvimento do *framework*, os autores não chegaram a trabalhar modelos detalhados de defeitos. Por esta razão, a versão atual do Neko possui um modelo bastante restrito, isto é, não existe ainda uma estrutura abrangente de simulação de defeitos que enquadre todo um modelo dentre os definidos na literatura, já descritos no capítulo 2.

Uma classe chamada `CrashEmulator` (Figura 3.7) proposta para simular colapso de processos, retém as mensagens de acordo com o tempo determinado para a permanência do defeito. Após o tempo de duração do colapso, na fase de recuperação (*recovery*), todas as mensagens retidas são enviadas. Neste modelo, um colapso (tal como denominado pelos autores) significa efetivamente que o processo foi temporariamente desconectado da rede. No entanto, o processo em colapso não deixa de realizar processamento e continua a enviar mensagens, embora estas mensagens não sejam recebidas pelo processo destino durante o período de defeito.

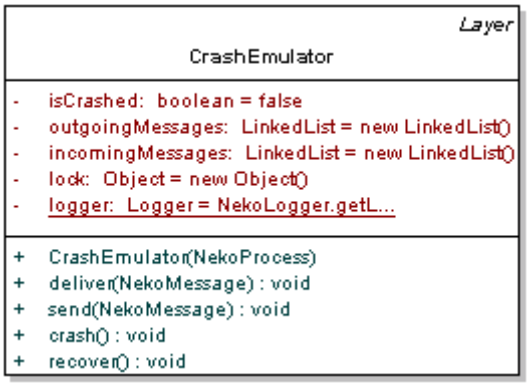


Figura 3.7: Diagrama da classe `lse.neko.layers.CrashEmulator`

Existem ainda soluções para a detecção de defeitos. A classe `FailureDetector` (Figura 3.8), por exemplo, mantém uma lista dos processos suspeitos (*suspected*), cuja instância deverá ser instalada em cada processo.

Para detecção de defeitos existe ainda a classe `Heartbeat`, além de outras incluídas no pacote `lse.neko.failureDetectors`.

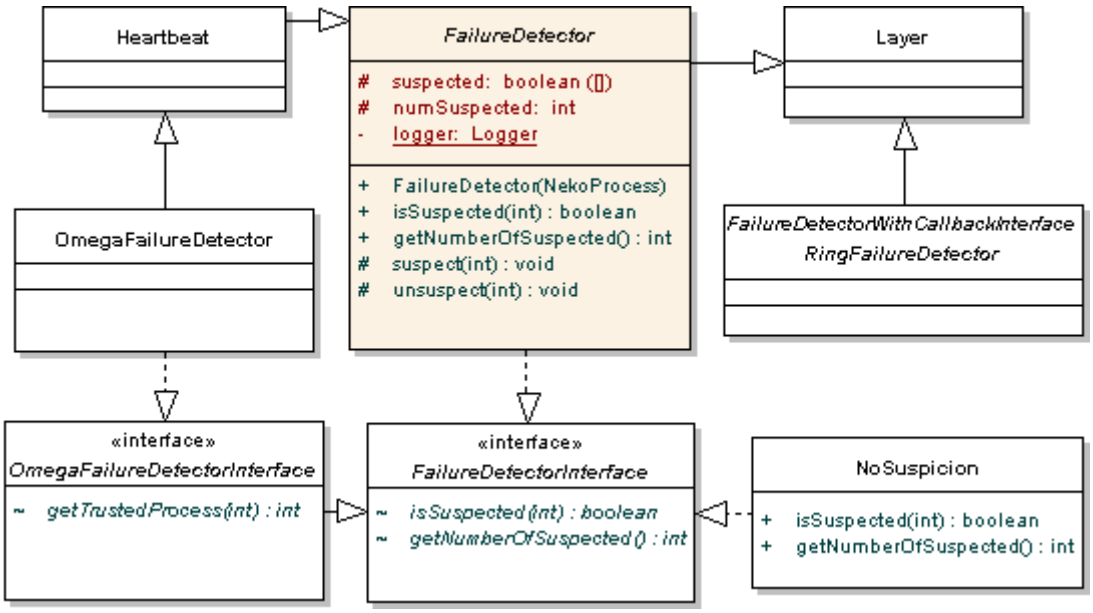


Figura 3.8: Detectores de defeitos implementados no Neko

3.4 Configuração e Execução das Simulações

Uma vez desenvolvida a camada de aplicação que será simulada, deve-se criar a estrutura de simulação que será utilizada. No Neko a simulação é configurada através de um arquivo próprio, que informa os parâmetros ao simulador, incluindo aqueles que o usuário deseje passar ao algoritmo que será simulado. Na Figura 3.9, é reproduzida parte do arquivo de configuração do exemplo disponível no Neko (`...\lse\neko\examples\basic`). Na linha 2 é especificado que o experimento será simulado, através do parâmetro *simulation*. Para emulação, este parâmetro deve receber o valor *false*. Na linha 5 é definido o número de processos participantes da simulação e a linha 8 informa a classe que irá iniciar a pilha de camadas em cada processo, chamado aqui de arquivo de inicialização. Neste exemplo todos os processos adotam o mesmo

arquivo de inicialização. Entretanto, é possível atribuir arquivos diferentes para cada processo, bastando para tanto informar o identificador do processo, no formato `process.<id>.initializer`. Os identificadores de processo podem possuir valores entre [0,*n*-1], onde *n* é o número de processos. As linhas 11-13 definem a rede que será utilizada. Neste caso, trata-se de uma rede simulada que recebe dois parâmetros (linhas 13 e 14). As linhas 16 à 24 definem as opções de *log*, incluindo o nome do arquivo de saída (linha 19) e a granulosidade[2] do *log*, isto é, quais informações devem ser registradas (linha 24). Por fim, na linha 27 é informado um parâmetro que define a classe que será adicionada como camada de aplicação. Esta classe será instanciada no arquivo de inicialização (Figura 3.10).

Uma vez criado o arquivo de configuração, o mesmo é passado como parâmetro para a classe de inicialização, que deve ser escrita em Java e estende a classe `lse.neko.NekoProcessesInitializer`. Nesta classe de inicialização, deve ser definido no método `init()` a estrutura de camadas que cada processo irá empregar. Portanto, cada processo deve possuir uma classe de inicialização. Para adicionar uma camada, utiliza-se o método `addLayer(<camada>)`, disponível na classe `lse.neko.NekoProcess`, onde `<camada>` é um objeto da classe que se deseja incluir. No exemplo da Figura 3.10, são adicionadas duas camadas. A primeira (linha 19) corresponde a uma camada que transforma as mensagens em *broadcast* em mensagens ponto-a-ponto. A segunda (linha 21) trata-se da camada de aplicação, que simula o algoritmo de consenso de Lamport (1978).

Qualquer camada pode ser incluída na pilha, desde que estenda a classe `lse.neko.Layer` ou `lse.neko.ActiveLayer` ou ainda que implemente a interface `lse.neko.LayerInterface`, herdando os métodos `send`, `receive` e `deliver`, necessários para transmitir as mensagens entre duas camadas distintas.

[2] A granulosidade do *log* indica o nível de detalhes em que as informações são registradas no arquivo. As mensagens trocadas entre os processos, por exemplo, são registradas no nível FINE. Outros níveis disponíveis são FINEST, INFO e HIGHER.

```
# Indicates that this is a simulation.
simulation = true

# The number of communicating processes.
process.num = 10

# The class that initializes the protocol stack of each process.
process.initializer = lse.neko.examples.basic.TestInitializer

# The network used for communication.
network = lse.neko.networks.sim.MetricNetwork
network.lambda = 1
network.multicast = true

#Options to enable the logging of messages into the file log.log.
handlers =
   java.util.logging.FileHandler,java.util.logging.ConsoleHandler
# Sets the name of the log file.
java.util.logging.FileHandler.pattern = log.log
# The messages channel logs all messages exchanged among processes
# at the FINE level.
# The console handler does not log them, as it only logs messages
# at levels INFO or higher.
messages.level = FINE

# Application parameter read by the TestInitializer class.
algorithm = lse.neko.examples.basic.Lamport
```

Figura 3.9: Exemplo de um arquivo de configuração da simulação

```
// lse.neko imports:
import lse.neko.LayerInterface;
import lse.neko.NekoProcess;
import lse.neko.NekoProcessInitializer;
import lse.neko.layers.NoMulticastLayer;

// other imports:
import org.apache.java.util.Configurations;

public class TestInitializer
   implements NekoProcessInitializer
{
   public void init(NekoProcess process, Configurations config)
       throws Exception
   {
       Class algorithmClass =
       Class.forName(config.getString("algorithm"));
       Class[] constructorParamClasses = { NekoProcess.class };
       Object[] constructorParams = { process };
       LayerInterface algorithm = (LayerInterface)
          algorithmClass
              .getConstructor(constructorParamClasses)
              .newInstance(constructorParams);
       process.addLayer(new NoMulticastLayer(process));
       process.addLayer(algorithm);
   }
}
```

Figura 3.10: Exemplo de um arquivo de inicialização da pilha de camadas

Para executar a simulação, deve ser utilizando o comando:

```
java lse.neko.Main arquivo_de_configuração.config
```

O arquivo com extensão `.config` é o que contém as informações de configuração da simulação.

Para executar o exemplo em uma rede real, são necessárias algumas modificações no arquivo de configuração. Entretanto, nenhuma alteração precisa ser feita nas demais classes definidas anteriormente. A Figura 3.11 descreve o código necessário para configurar uma emulação utilizando uma rede real. O parâmetro *simulation* deve receber o valor `false` (linha 2), indicando que não se trata de uma simulação. É necessário também definir os endereços das máquinas onde os processos serão executados. Estes endereços podem ser os IPs ou os endereços de domínio que seguem o padrão da rede em uso. Como neste exemplo existem três processos, é necessário definir dois endereços. O processo 0 (zero) será executado na máquina *home* e os processos 1 e 2 serão executados nas máquinas `slave`. Por fim, o parâmetro `network` precisa receber uma classe de rede real válida, como `TCPNetwork` ou `UDPNetwork`, já discutidas na seção 3.2.

Para iniciar a emulação é necessário primeiramente executar o comando:

```
java lse.neko.comm.Slave
```

em cada máquina *slave* e, só então iniciar a execução do algoritmo na máquina *home*. Isto é necessário para permitir que o Neko consiga instanciar as classes nas máquinas remotas.

```
1  # Indicate that this is a simulation.
2  simulation = false
3
4  # The number of communicating processes.
5  process.num = 3
6
7  # Options to find the other processes on the network.
8  slave = host1.your.net,host2.your.net
9
10 # The class that initializes the protocol stack of each process.
11 process.initializer = lse.neko.examples.basic.TestInitializer
12
13 # The network used for communication.
14  network = lse.neko.networks.comm.TCPNetwork
15
16 #Options to enable the logging of msgs into the file log.log.
17 handlers = java.util.logging.FileHandler,
18            java.util.logging.ConsoleHandler
19 # Sets the name of the log file.
20 java.util.logging.FileHandler.pattern = log.log
21 # The message channel logs all msgs exchanged among processes
22 # at the FINE level.
23 # The console handler does not log them; it only logs msgs
24 # at levels INFO or HIGHER.
25 messages.level = FINE
26
27 # Application parameter read by the TestInitializer class.
28 algorithm = lse.neko.examples.basic.Lamport
```

Figura 3.11: Exemplo de arquivo de configuração para execução em uma rede real

Com todos os mecanismos descritos neste capítulo conclui-se que o Neko possui um suporte para simulação e emulação de algoritmos distribuídos bastante amplo e bem elaborado. Além disso, a possibilidade de inserção de camadas é, sem dúvida, favorável para a inserção de novas funcionalidades na ferramenta. O suporte a simulação de defeitos é ainda bastante restrito, o que incentiva e justifica a pesquisa realizada neste trabalho.

4 MODELAGEM E PROPOSTA DE SOLUÇÃO PARA A SIMULAÇÃO DE DEFEITOS NO NEKO

Seguindo as definições de defeitos apresentadas no capítulo 2, foram desenvolvidas propostas para simulação de defeitos de omissão, colapso, rede e particionamento. Além disso, foram feitas observações sobre a simulação dos defeitos de temporização e resposta, embora estes não tenham sido incluídos nas implementações.

Os defeitos de omissão foram divididos em omissão no envio, omissão no recebimento e omissão geral. Os defeitos de colapso incluem colapso por parada, por pausa, com amnésia total e com amnésia parcial. Além disso, foram ainda propostos quatro modelos de descarte para serem utilizados em conjunto com os defeitos de omissão e rede.

Neste capítulo, estão descritos apenas os modelos sugeridos para a geração dos tipos de defeitos estudados. Os detalhes e ajustes efetuados durante a implementação estão descritos no capítulo 5.

4.1 Defeitos de Omissão

Os defeitos de omissão foram divididos em duas classes principais: omissão no envio (*send omission*) e omissão no recebimento (*receive omission*). Estes defeitos podem ocorrer por falta de espaço no buffer do sistema operacional ou na interface de rede, ocasionando o descarte de mensagens (BIRMAN, 1996). Os defeitos de omissão geral caracterizam-se pela geração simultânea de omissões no envio e no recebimento.

A estrutura necessária para simulação dos defeitos de omissão no envio pode ser obtida com a inserção de uma nova camada na pilha de camadas do processo que estará sujeito a este tipo de defeito (Figura 4.1). Esta deverá ser adicionada entre as camadas `NekoProcess` e a aplicação, de acordo com a necessidade da aplicação em teste. A partir desta inserção, todas as mensagens provenientes das camadas acima daquela inserida para causar omissão serão interceptadas e estarão sujeitas ao descarte.

A proposta para simulação dos defeitos de omissão no recebimento segue a mesma estrutura utilizada para a omissão no envio. A sugestão é construir uma camada que intercepte as mensagens recebidas (Figura 4.2) e faça o descarte de acordo com uma política, registrando o descarte no arquivo de *log*. Igualmente ao descarte por omissão no envio, a política poderá ser definida pelo usuário através de mecanismo auxiliar como, por exemplo, o arquivo de configuração.

Numa abordagem inicial deste trabalho (RODRIGUES, 2005), as mensagens eram descartadas de acordo com uma porcentagem, que deveria ser informada no arquivo de configuração. Entretanto, visando prover uma maior flexibilidade, optou-se por criar um mecanismo que permita ao usuário do Neko utilizar uma política de descarte já disponível ou desenvolver a sua própria política de descarte e informá-la no início da simulação. As políticas de descarte disponibilizadas estão descritas na seção 4.5.

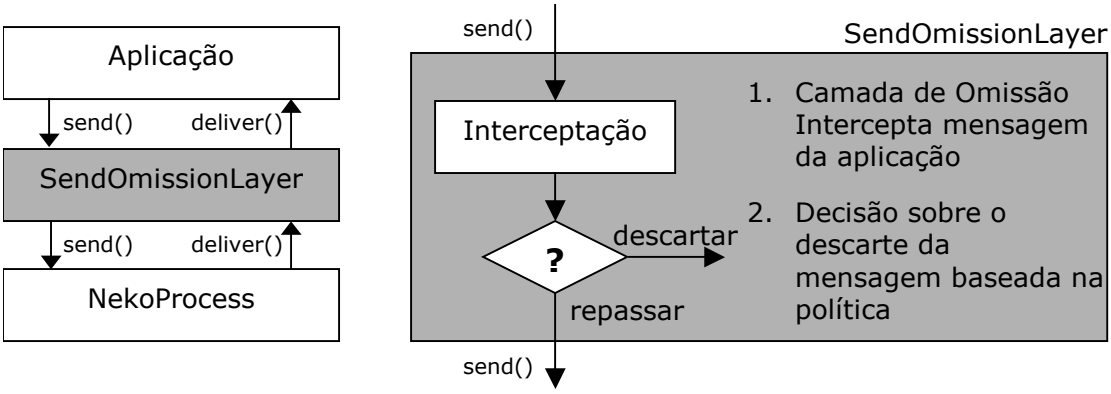


Figura 4.1: Estrutura para descarte por omissão no envio

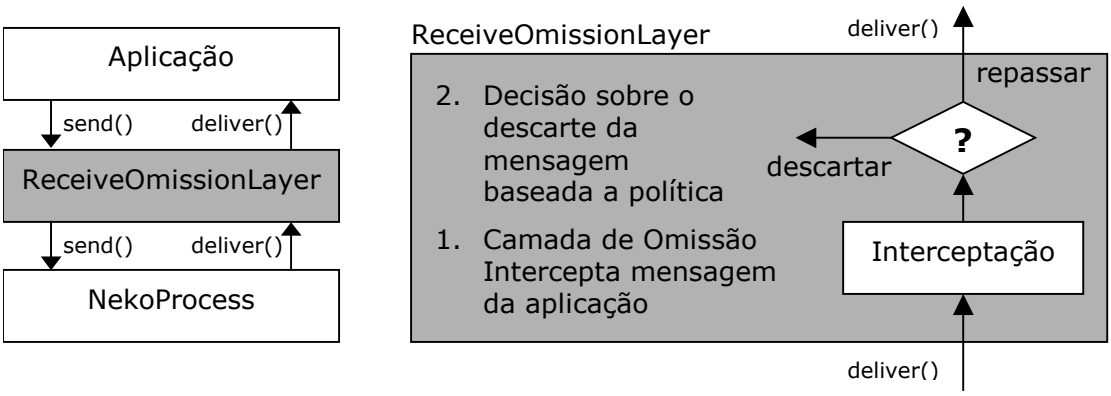


Figura 4.2: Estrutura para descarte por omissão no recebimento

Sendo modelada como uma camada, é possível inserir a camada de omissão em qualquer posição da pilha de camadas do processo. Entretanto, deve-se levar em consideração que a camada de descarte no envio atua sobre as mensagens enviadas pelas camadas superiores, o que significa que, mensagens enviadas abaixo desta camada não sofrerão descarte. O mesmo acontece com a camada de descarte no recebimento, que atua somente sobre as mensagens que serão recebidas pelas camadas superiores. As mensagens recebidas abaixo desta camada não serão incluídas no descarte. Isto deve ser utilizado convenientemente, de acordo com a necessidade do experimento. De qualquer forma, para incluir todas as mensagens no cálculo de descarte (seja no envio ou no recebimento) deve-se incluir as respectivas camadas no nível mais inferior, logo acima da camada `NekoProcess`. Isto é válido para mensagens trocadas entre processos. Embora não seja o mais comum, é possível que uma camada envie mensagens para outra camada no mesmo processo. Neste caso, fica a cargo do usuário determinar a melhor localização para as camadas de omissão.

Para modelar o defeito de omissão geral, proposto por Neiger e Toueg (1988), basta incluir as camadas de omissão no envio e no recebimento simultaneamente na pilha de camadas, sempre levando em consideração os cuidados sugeridos no parágrafo anterior.

A configuração dos defeitos de omissão pode ser realizada definindo-se pares de intervalos de tempo (ti_i, tf_i), representando o início e fim do defeito, respectivamente. Caso estes tempos sejam omitidos, as mensagens de aplicação estarão sujeitas a omissão do início ao fim da simulação. Opcionalmente pode-se definir apenas o início para a ocorrência dos defeitos. Neste caso, os defeitos ocorrerão a partir do tempo inicial até o final da simulação. Definir um tempo inicial é bastante útil quando se deseja evitar que o defeito ocorra no início da simulação, quando o sistema pode estar em fase de carga.

Da mesma forma, o final da simulação pode necessitar de mensagens de sincronização, que serão indesejavelmente afetadas com a ocorrência de omissões se estas não forem inibidas.

4.2 Defeitos de Colapso

De acordo com Cristian (1991), os defeitos de colapso (*crash failures*) ocorrem quando o servidor pára de responder, podendo ou não ser reinicializado, manifestando comportamentos que podem ser descritos através de quatro subclasses: colapso por pausa, por parada, com amnésia parcial e com amnésia total. Referente aos defeitos de parada, Birman (1996) ainda acrescenta que os mesmos ocorrem quando um processo ou computador pára sua execução sem realizar qualquer ação incorreta. Neste caso, a única maneira de se deduzir que o sistema parou é através de *timeouts*, seja pela ausência de mensagens "*keep alive*", ou seja, pela ausência de respostas às mensagens de *ping*.

Visando disponibilizar uma estrutura básica para a simulação dos defeitos de colapso de acordo com as definições acima, optou-se por desenvolver uma camada que possa ser adicionada na pilha de camadas de cada processo (Figura 4.3). Esta camada fará a interceptação das mensagens enviadas e recebidas e irá descartá-las caso o processo esteja em colapso. O comportamento do processo pode variar de acordo com o tipo de colapso utilizado.

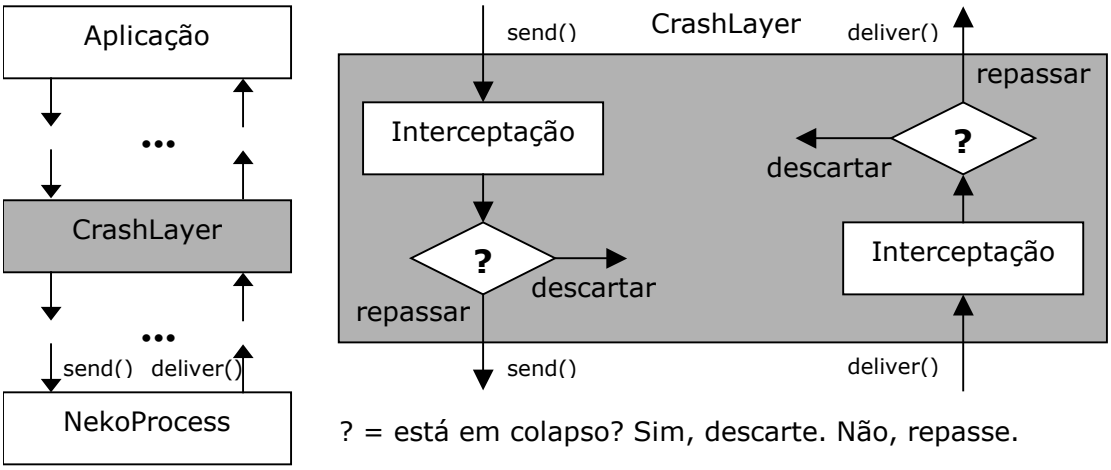


Figura 4.3: Estrutura para simulação de Colapso no Neko

A camada de colapso, designada `CrashLayer`, deve conter apenas as primitivas básicas para geração de colapsos. Estas primitivas devem incluir métodos para sinalizar o início e o fim do defeito. Outra funcionalidade da camada `CrashLayer` é o registro de descarte de todas as mensagens que deveriam ser enviadas e recebidas durante o colapso.

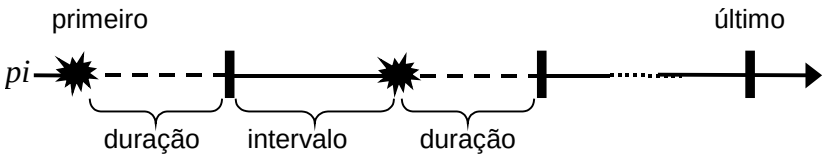
No colapso por pausa, a aplicação que está sendo executada é pausada no momento da ocorrência do defeito. Caso a mesma seja reiniciada, sua execução continuará a partir do ponto em que parou.

O colapso por parada acontece quando o processo pára sua execução, não havendo retorno posterior. Este tipo de defeito pode ser visto como um colapso com pausa infinita.

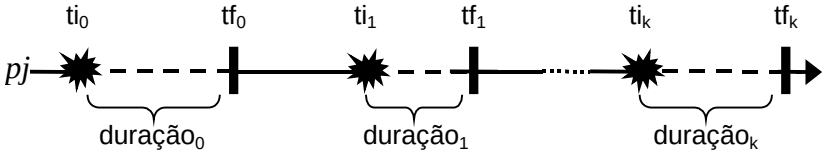
O colapso com amnésia pode exibir um dos seguintes comportamentos: parcial ou total. O processo que sofre amnésia parcial retorna a um estado intermediário, que pode ser representado pelo último estado salvo antes do colapso. Este comportamento é

diferente do colapso por pausa, visto que o processo pode ter evoluído na computação desde o último salvamento de estado, o que não ocorre no colapso por pausa. No caso de uma amnésia total, a aplicação irá voltar ao estado inicial e toda a computação anterior será perdida.

A configuração proposta para a ocorrência destes defeitos pode ser realizada de duas formas alternativas (Figura 4.4). Na configuração A, os colapsos ocorrem em intervalos regulares com duração constante a partir de um tempo inicial, referente a primeira ocorrência do defeito. Os defeitos ocorrem em intervalos fixos até que o experimento termine. Pode-se ainda determinar um intervalo de tempo para que os defeitos cessem, representado pelo parâmetro "último" da figura. Este parâmetro não representa uma ocorrência do defeito: ele apenas marca o intervalo de tempo a partir do qual não ocorrerão mais defeitos.



(a) Configuração A



(b) Configuração B

Figura 4.4: Configuração para a ocorrência de colapso

A segunda opção (configuração B) é definir tempos fixos, porém com intervalo e duração variáveis, analogamente ao proposto nos defeitos de omissão. Neste caso devem ser definidos os tempos iniciais e finais de cada ocorrência do defeito. Opcionalmente pode-se definir um tempo inicial ti_k que não possua o tempo final tf_k correspondente. Neste caso, o defeito permanecerá até o final do experimento.

O defeito de colapso por parada é o único que necessita apenas do tempo de início de sua ocorrência, visto que o mesmo ocorre uma única vez e não existe retorno. Este tipo de defeito também pode ser obtido considerando um colapso com pausa, desde que o intervalo de duração seja grande o suficiente para garantir que a aplicação não retorne do defeito durante o tempo restante de simulação.

Da mesma forma como ocorre na solução de omissão, a posição da camada de colapso pode influenciar o comportamento do experimento. As mensagens trocadas abaixo da camada de colapso serão enviadas e recebidas normalmente. Para isolar todas as camadas da pilha, isto é, descartar todas as mensagens enviadas e recebidas, basta adicionar a camada de colapso logo acima da camada `NekoProcess`.

4.3 Defeitos de Rede e Particionamento

Como já discutido na seção 3.2, até a versão 0.8 do Neko, todos os processos em uma rede simulada eram conectados diretamente. Embora na nova versão (0.9) tenham sido adicionados mecanismos para incluir topologia, as redes da versão 0.8 continuam

disponíveis e podem ser utilizadas em casos mais simples ou quando a topologia não é necessária. Por esta razão, decidiu-se criar soluções para geração de defeitos nos modelos de rede da versão 0.8. Foram ainda levantados os mecanismos disponíveis no SSFNet que possibilitam a geração de defeitos no nível de rede, embora não se tenha realizado nenhuma alteração neste simulador.

Para evitar a alteração de todas as classes de rede implementadas no Neko (seção 3.2), optou-se por criar uma rede com suporte a defeitos que recebe como parâmetro uma outra rede, dentre as disponíveis no *framework*. Desta forma, a rede com defeitos utiliza os mecanismos de envio das redes já desenvolvidas, dispensando qualquer alteração nas mesmas (Figura 4.5).

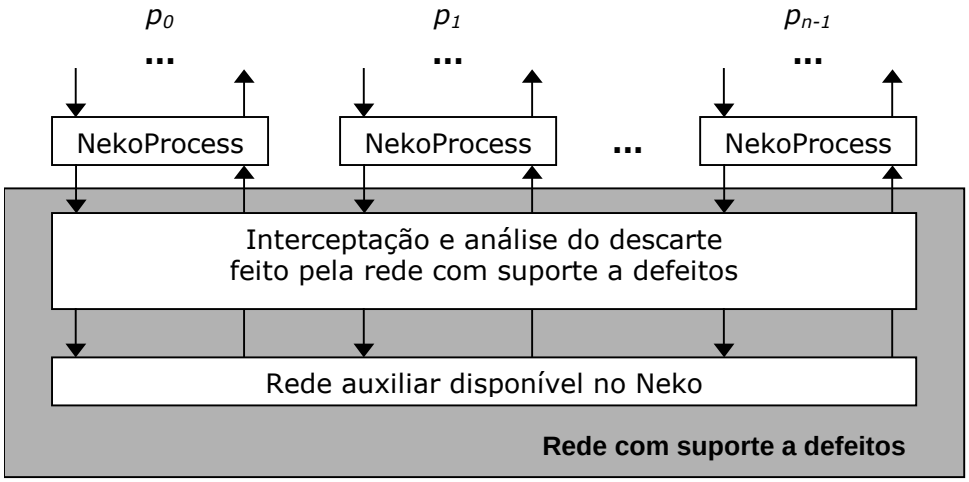
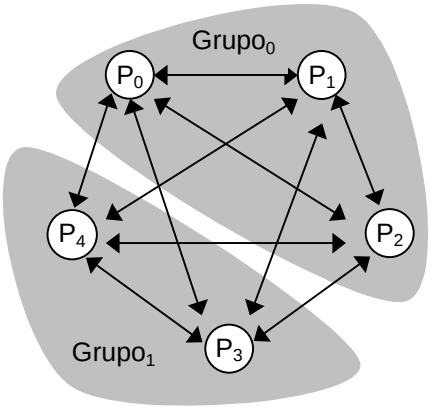


Figura 4.5: Arquitetura para inserção de defeitos de rede e particionamento

Um dos defeitos de rede modelados efetua descarte de mensagens de acordo com a política escolhida. As políticas são as mesmas utilizadas pelos defeitos de omissão, que serão descritas na seção 4.5.

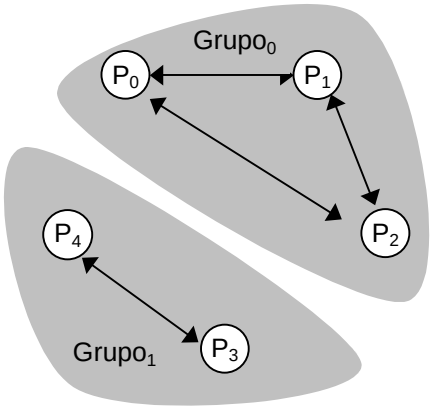
O segundo tipo de defeito modelado é a desconexão ou quebra do enlace entre dois processos. O modelo de rede com defeito mantém uma matriz de ligação entre processos com o estado do enlace. Toda vez que uma mensagem é enviada para a rede é verificado se o enlace está em estado UP ou DOWN. Se o enlace está UP, a mensagem é enviada normalmente. Se o enlace está DOWN, a mensagem é descartada.

Fazendo uso dos mecanismos de configuração do estado dos enlaces, modelou-se também uma rede com defeitos que permite simular o particionamento dos nodos da rede. Neste caso, a solução prevê a criação de grupos de processos na rede, ao quais corresponderão às futuras partições. Como exemplo, suponha-se que existam 5 processos na simulação e que os processos P_0, P_1 e P_2 estejam em um grupo ($Grupo_0$) e os processos P_3 e P_4 em outro grupo ($Grupo_1$), como ilustrado na Figura 4.6. Quando o particionamento do $Grupo_0$ foi solicitado, todas as conexões entre os processos do $Grupo_0$ com os processos restantes (processos P_3 e P_4) serão interrompidas. Isto significa que todas as mensagens trocadas entre estes dois grupos serão descartadas.



	P_0	P_1	P_2	P_3	P_4
P_0	-	UP	UP	UP	UP
P_1	UP	-	UP	UP	UP
P_2	UP	UP	-	UP	UP
P_3	UP	UP	UP	-	UP
P_4	UP	UP	UP	UP	-

(a) antes do particionamento



	P_0	P_1	P_2	P_3	P_4
P_0	-	UP	UP	DOWN	DOWN
P_1	UP	-	UP	DOWN	DOWN
P_2	UP	UP	-	DOWN	DOWN
P_3	DOWN	DOWN	DOWN	-	UP
P_4	DOWN	DOWN	DOWN	UP	-

(b) depois do particionamento

Figura 4.6: Exemplo de particionamento

As soluções anteriores não incluem os novos mecanismos para criação de topologias mais complexas, disponibilizados com a integração do SSFNet com a nova versão do Neko (versão 0.9). Como esta integração foi realizada por terceiros e divulgada recentemente (MATSUSHITA, 2005) seria inviável, principalmente por razões de tempo, realizar um estudo mais detalhado do SSFNet e propor alterações para simulação de defeitos de rede no mesmo. Neste sentido, optou-se levantar quais são os mecanismos de simulação de defeitos disponíveis no SSFNet e como estes podem ser utilizados dentro do escopo deste trabalho.

Através de uma análise superficial do simulador foi encontrado um mecanismo que permite a configuração da taxa de descarte de pacotes na interface de comunicação dos nodos da rede. Cada nodo, seja roteador ou *host*, possui uma ou mais interfaces de comunicação, utilizadas para fazer a ligação com outros nodos. Toda interface possui um parâmetro de configuração chamado `flaky` (Figura 4.7). Este atributo configura o descarte de pacotes com uma probabilidade p, onde $0.0 \leq p \leq 1.0$. Na ausência do atributo `flaky` o valor 0.0 é escolhido como padrão.

```
interface [
        id %I
        idrange [from %I1! to %I1!]
        ip %S

        bitrate %F
        latency %F

        ...
        flaky   %F
]
```

Figura 4.7: Parâmetros de configuração de interfaces no SSFNet

4.4 Demais Classes de Defeitos

Seguindo as classes propostas por Cristian (1991) e Birman (1996), descritas no capítulo 2, resta ainda propor soluções para simular defeitos de temporização (*timming failure*), resposta (*response failure*) e bizantinos.

A classe de temporização possui uma implementação chamada *LeakyBucket* (`lse.neko.layers.LeakyBucket`) que estende a classe `lse.neko.Layer`. Esta camada impõe uma taxa máxima de envio de mensagens através dela. As mensagens são armazenadas em um *buffer* e são repassadas para a próxima camada apenas quando o atraso mínimo é alcançado, garantindo um tempo mínimo entre dois envios.

Os defeitos de resposta não possuem implementação disponível na versão atual. Entretanto, seguindo a mesma linha das implementações dos defeitos de omissão, uma solução seria criar uma camada que interceptasse e alterasse o conteúdo das mensagens.

No entanto, estas classes de defeitos não serão abordados neste trabalho por não haver tempo suficiente para realizar um estudo aprofundado sobre o comportamento destes tipos de defeitos, incluindo as propostas para simular a ocorrência dos mesmos, implementação, testes e validação das possíveis soluções. Além disso, os defeitos bizantinos são difíceis de tratar devido a alta complexidade de modelagem do mesmo.

4.5 Modelos de Descarte

Existem diversos modelos de descarte disponíveis na literatura, tais como os descritos por Jiang e Schulzrinne (2000) e Markovski (2000). Alguns deles surgiram ainda na década de 60, como uma tentativa de modelar o comportamento dos erros nos canais de telefonia. Dentre os modelos mais utilizados destacam-se: o de Bernoulli ou *Memoryless*, o *two-state* de Markov ou modelo de Gilbert e suas extensões e o *multi-state* de Markov. Existem ainda abordagens que utilizam distribuições estatísticas, como o modelo de Mertz que utiliza distribuições hiperbólicas, e o de Berger e Mandelbrot, que utiliza a distribuição de Pareto (VOIP, 2005).

Uma vez que a implementação de modelos de descarte não é o objetivo principal deste trabalho, foram disponibilizados a título de exemplificação, quatro soluções: Bernoulli, Gilbert e duas outras alternativas mais simples, onde uma utiliza uma lista de mensagens a serem descartadas, e outra gera descartes em intervalos periódicos. Estes quatro modelos também são utilizados no NS (*Network Simulator*) (NS, 2005).

Para facilitar o entendimento dos modelos utilizados, os mesmos serão discutidos mais detalhadamente a seguir. A inserção de outras soluções de descarte no Neko está incluída nos trabalhos futuros.

4.5.1 Modelo de Bernoulli

O funcionamento do modelo de Bernoulli é baseado em um único parâmetro, que representa a taxa de descarte "*r*" do processo (MARKOVSKI, 2000). Todas as mensagens que são interceptadas recebem um número aleatório e o descarte é feito de acordo com este número. Suponha-se, por exemplo, que a taxa de descarte esteja no intervalo [0, 1], o que corresponde a uma taxa que varia de 0% a 100%. Se a taxa *r* for de 0,1 (10%) serão gerados números aleatórios entre 0 e 1 e todas as mensagens que receberem números menores que 0,1 serão descartadas. O mesmo acontece para qualquer outro valor de *r*. Desta forma, para uma seqüência de *N* mensagens, o número de descartes tende a *N***r*, para valores grandes de *N*.

A vantagem deste modelo é que não há necessidade de análise das mensagens anteriores para efetuar o descarte. Entretanto, como as mensagens são descartadas

aleatoriamente, não há garantia de que as mensagens descartadas correspondam exatamente à taxa de descarte informada, nem que os processos descartarão a mesma quantidade de mensagens.

4.5.2 Modelo de Gilbert

O modelo *two-state* de Markov ou modelo de Gilbert (JIANG e SCHULZRINNE, 2000) permite a geração de descartes em rajada (*burst*). O modelo é baseado em dois estados (Figura 4.8). O estado "1" representa o descarte de mensagens e o estado "0" representa o envio correto das mensagens. *p* é a probabilidade de transição do estado 0 para o estado 1 e *q* é a probabilidade de transição do estado 1 para o estado 0.

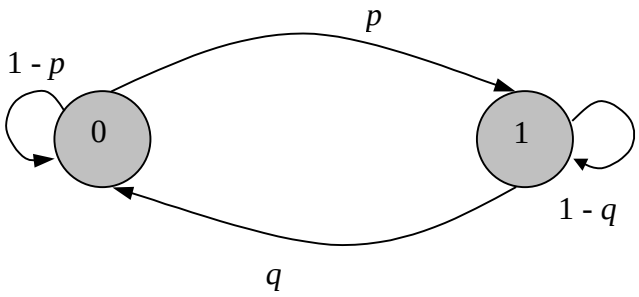


Figura 4.8: Modelo de Gilbert

Neste modelo, a taxa de descartes *r* é dada pela equação (1). A probabilidade *q* está relacionada à "seqüencialidade" do descarte, isto é, ao tamanho da rajada. A probabilidade *pk* de ocorrer uma rajada de tamanho *k* pode ser obtida através da equação (2). O descarte em rajada pode ser modelado escolhendo-se apropriadamente valores para *p* e *q* ou alternativamente os de *r* e *q*.

$$r = \frac{p}{(p+q)} \qquad (1) \qquad\qquad pk = q \cdot (1-q)^{(k-1)} \qquad (2)$$

É válido observar que, quando *q* é igual a 1-*p*, o modelo de Gilbert se resume ao modelo de Bernoulli.

Para evitar algumas situações particulares, Trumbo (2005) recomenda que os valores de *p* e *q* devem estar estritamente entre 0 e 1, com $0 < p, q < 1$. Quando esta condição não é satisfeita, podem ocorrer os seguintes comportamentos:

- $p = q = 1$: possui um comportamento conhecido como "*flip-flop*". A cada passo o estado do autômato é alternado entre 0 e 1;
- $p = q = 0$: seja qual for o estado inicial, o mesmo será mantido em todos os passos subseqüentes.
- $p = 0$, $0 < q < 1$: se o processo inicia no estado 1, existe uma probabilidade *q* de que, em algum passo, ocorrerá uma transição para o estado 0. Uma vez no estado 0, o mesmo será mantido em todos os demais passos. Quando isto ocorre, é dito que o processo foi absorvido no estado 0. O número de passos até que ocorra a absorção possui uma distribuição geométrica com média $1/q$. Obviamente, se o processo inicia no estado 0 ele é absorvido desde o início;
- $q = 0$, $0 < p < 1$: o comportamento é o inverso do anterior, com a absorção ocorrendo no estado 1.

4.5.3 Lista de descarte

Os dois modelos anteriores efetuam descartes de forma não determinística e, por esta razão, são mais condizentes com a realidade. Entretanto, podem existir casos em que o descarte determinístico seja útil, principalmente quando se deseja incluir descartes em pontos estratégicos. Neste sentido, propõe-se criar um modelo de erros para efetuar descartes de acordo com uma lista que contém os números de seqüência das mensagens que serão descartadas. Por exemplo, se a lista for composta pelos números [28, 29, 40, 42], serão descartadas a 28ª, 29ª, 40ª e 42ª mensagem, de acordo com a ordem de processamento da mensagem. Este tipo de modelo é útil quando se conhece de antemão o número de mensagens que serão enviadas e quais delas devem ser descartadas para efetuar o teste da aplicação.

4.5.4 Descarte em intervalos periódicos

Um outro modelo que pode ser utilizado quando se deseja efetuar descartes de forma determinística é o descarte em intervalos periódicos. A cada *n*-ésima mensagem transmitida, a última é descartada. Este modelo é útil quando não se conhece de antemão o tempo total para a execução do experimento e, portanto, não se pode determinar com precisão a lista de mensagens, como no modelo anterior. Além disso, algumas vezes pode ser inviável definir uma lista de mensagens quando o número de mensagens a serem descartas é grande.

5 IMPLEMENTAÇÃO

No capítulo 4, foram definidos os modelos que este trabalho propõe para desenvolver a infra-estrutura de simulação de defeitos no Neko. O presente capítulo apresenta os aspectos relacionados à implementação das estruturas e os mecanismos necessários para atender a modelagem proposta. São descritas as classes Java geradas e a relação entre elas, bem como as alterações necessárias nas classes Java já disponíveis no Neko. Além disso, para soluções que utilizam parâmetros de entrada, foi especificada a sintaxe dos mesmos.

Primeiramente são apresentados os defeitos de omissão, incluindo omissão no envio, omissão no recebimento e omissão geral. Em seguida, é apresentada a implementação dos defeitos de colapso por parada, por pausa e por amnésia. Por último, segue a descrição da implementação dos defeitos de rede e particionamento.

5.1 Defeitos de Omissão

Os defeitos de omissão causam o descarte de mensagens de acordo com a necessidade da aplicação. A implementação destes tipos de defeitos foi dividida em duas classes Java distintas: `SendOmissionLayer` e `ReceiveOmissionLayer`, modelando os defeitos de omissão no envio e no recebimento, respectivamente. É possível utilizar as duas classes em conjunto, caracterizando os defeitos de omissão geral, já discutidos nas seções anteriores.

5.1.1 Omissão no envio

A estrutura necessária para simulação dos defeitos de omissão no envio prevê a inserção de uma nova camada na pilha de camadas do processo que estará sujeito a este tipo de defeito. Esta camada foi modelada em uma classe Java nomeada `SendOmissionLayer` (Figura 5.1).

Para interceptar as mensagens, a classe `SendOmissionLayer` sobrescreve o método `send()` herdado da classe `lse.neko.Layer`. Este método é invocado toda vez que a camada superior envia uma mensagem. É possível informar os intervalos de tempo para início e fim da ocorrência deste defeito. Os parâmetros disponibilizados são:

`process.<id>.sendomission.start = <ti`$_0$`>,<ti`$_1$`>, ..., <ti`$_n$`>`
`process.<id>.sendomission.finish = <tf`$_0$`>, <tf`$_1$`>, ..., <tf`$_m$`>`

É possível definir tantos intervalos quantos forem desejados, bastando incluir para cada *ti* um *tf* respectivo. Pode-se ainda definir um último *ti* que não possua *tf*. Neste caso o defeito será mantido até o final do experimento. Da mesma forma, pode-se definir apenas o início do defeito, suprimindo-se o parâmetro `finish`. Para

experimentos onde a omissão deve ocorrer do início ao fim, basta suprimir os parâmetros `start` e `finish`.

O descarte também pode ser feito de acordo com uma política definida pelo usuário. Para utilizar a sua política, o usuário deve criar uma classe que implemente a interface `DropModelInterface` e adicionar ao método `drop()` o critério para descarte. A classe `SendOmissionLayer` invocará este método toda vez que uma mensagem for recebida. Se o método retornar `true`, a mensagem será descartada. Caso contrário, a mensagem será repassada para a camada inferior.

Para informar à classe `SendOmissionLayer` qual é a classe que implementará a política de descarte, deve-se adicionar ao arquivo de configuração o parâmetro:

```
process.sendomission.dropmodel = <política>
```

A `<política>` corresponde ao nome da classe que implementa a política de descarte. Os modelos disponibilizados, já discutidos na seção 4.5, foram mapeados nas classes `BernoulliDropModel`, `GilbertDropModel`, `ListErrorModel` e `PeriodicErrorModel`.

Ainda com relação à configuração da ocorrência do defeito de omissão, é possível definir políticas diferenciadas para cada processo, bastando informar o número identificador do processo `<id>` que utilizará a política, na forma:

```
process.<id>.sendomission.dropmodel = <política>
```

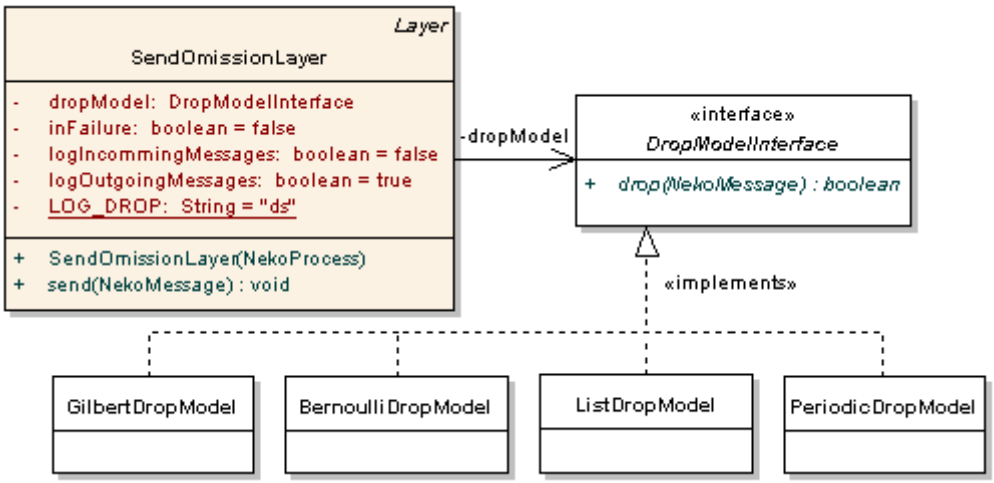


Figura 5.1: Diagrama da classe `lse.neko.layers.SendOmissionLayer`

Para adicionar a camada `SendOmissionLayer` ao processo, basta executar o procedimento padrão de criação da pilha de camadas. Cria-se uma classe que estenda a classe `lse.neko.NekoProcessInitializer` e no método `init()` inclui-se a linha de comando `process.addLayer (new SendOmissionLayer (process))`, não esquecendo de informar o parâmetro *process* (objeto identificador do processo).

No exemplo da Figura 5.2, são adicionadas duas camadas: a primeira de descarte de mensagens (linha 19) e a segunda de aplicação (linha 20). Na linha 4 está especificada a localização da classe `SendOmissionLayer` dentro do pacote Neko. A camada de rede não aparece neste arquivo porque é informada através do arquivo de configuração. Portanto, a primeira camada adicionada vem logo após a camada `NekoProcess`, e assim sucessivamente para as demais camadas até à camada de aplicação, ou seja, a pilha é montada de baixo para cima.

```
1  // lse.neko imports:
2  import lse.neko.NekoProcess;
3  import lse.neko.NekoProcessInitializer;
4  import lse.neko.layers.omission.SendOmissionLayer;
5
6  // other imports:
7  import org.apache.java.util.Configurations;
8
9  public class Initializer
10    implements NekoProcessInitializer
11 {
12
13   public Initializer() {
14   }
15
16   public void init(NekoProcess process, Configurations config)
17      throws Exception
18   {
19     process.addLayer(new SendOmissionLayer(process));
20     process.addLayer(new Lamport(process));
21   }
22 }
```

Figura 5.2: Exemplo de instanciação das camadas utilizando `SendOmissionLayer`

A camada `SendOmissionLayer` é também responsável pelo registro de descarte das mensagens. O *log* segue o padrão utilizado pelo Neko para os demais eventos registrados, como envio e recebimento de mensagens entre processos.

Para cada mensagem descartada é gerada uma linha no arquivo de *log* da simulação de acordo com um exemplo mostrado através da Tabela 5.1.

Cada linha contém as seguintes informações:

- col 1: instante de tempo em que ocorreu o evento de descarte;
- col 2: processo que originou o evento de descarte;
- col 3: identificador da classe (`SendOmissionLayer`);
- col 4: parâmetro fixo, que representa a ocorrência de um evento (*event*);
- col 5: indicador do tipo de evento. *ds* sinaliza evento de descarte por omissão no envio (*drop on send*);
- col 6: identificador do processo de origem da mensagem;
- col 7: identificador(es) do(s) processo(s) de destino da mensagem;
- col 8 e col 9: tipo e conteúdo da mensagem, respectivamente.

Tabela 5.1: Trecho do *log* gerado pela classe `SendOmissionLayer`

col 1	col 2	col 3	col 4	col 5	col 6	col 7	col 8	col 9
1,876	p1	messages	e	s	p1	p0	APP	null
1,876	P2	messages	e	s	P2	p0	APP	null
1,876	P3	messages	e	s	P3	p0	APP	null
4,876	p0	messages	e	r	p1	p0	APP	null
5,876	p0	messages	e	r	p2	p0	APP	null
6,752	p0	SendOmissionLayer	e	s	p0	p1	ACK	null
6,876	p0	messages	e	r	P2	p0	APP	null
8,628	p0	SendOmissionLayer	e	s	p0	p2	ACK	null
8,876	p0	messages	e	r	P3	p0	APP	null
10,504	p0	SendOmissionLayer	e	ds	p0	P2	ACK	null
12,380	p0	SendOmissionLayer	e	ds	p0	P3	ACK	null
...								

Um inconveniente pode surgir quando as camadas de omissão são utilizadas. Todas as mensagens enviadas e recebidas são registradas no arquivo de *log* pela camada `NekoProcess`, localizada logo acima da camada de rede. A camada de omissão é adicionada acima da camada `NekoProcess`. Isto significa que a mensagem, mesmo sendo descartada pela camada de omissão, pode ser registrada como recebida pelo processo, embora seu descarte seja registrado posteriormente. Para evitar este tipo de inconsistência e para que não seja necessário realizar um pós-processamento do *log* de simulação em busca deste tipo de ocorrência, foram definidos mecanismos para ativar e desativar o registro de mensagens na camada `NekoProcess`. A solução implementada consiste em utilizar duas variáveis booleanas para indicar se o registro deve ou não ser realizado nesta camada. Uma dessas variáveis, denominada `logIncomingMessages` é responsável por ativar ou desativar o registro das mensagens recebidas e a outra, denominada `logOutgoingMessages`, ativa ou desativa o registro das mensagens enviadas. Assim, fica a cargo da classe `SendOmissionLayer` efetuar o registro das mensagens enviadas e não descartadas. Da mesma forma, a classe `ReceiveOmissionLayer` fica responsável por registrar as mensagens recebidas e que não foram descartadas. Entretanto, para garantir a compatibilidade com as aplicações já desenvolvidas com o Neko, esta configuração deve ser explicita, isto é, a camada `NekoProcess` continua registrando todas as mensagens a não ser que seja explicitamente configurada para não fazê-lo.

Se o registro das mensagens for realizado pelas camadas de omissão, dependendo da localização na pilha de camadas do processo, mensagens podem não ser registradas no *log*. Isto é, se a camada de omissão no envio, por exemplo, estiver acima de uma outra camada que envie mensagens, estas mensagens não serão registradas. Assim, caso seja utilizado o *log* na camada de omissão, fica a cargo do usuário verificar este tipo de ocorrência e evitá-la, se necessário.

5.1.2 Omissão no recebimento

A classe que simula descartes por defeitos de recebimento é muito semelhante àquela que simula omissão no envio, descrita na seção anterior. Uma classe `ReceiveOmissionLayer` (Figura 5.3) estende a classe `lse.neko.Layer` e sobrescreve o método `deliver()`. As mensagens são interceptadas antes de serem entregues à camada superior e, de acordo com a política escolhida, são descartadas ou repassadas.

Analogamente ao defeito de omissão no envio, é possível definir os intervalos de início e término da ocorrência das omissões, seguindo os seguintes parâmetros:

```
process.<id>.receiveomission.start = <ti0>,<ti1>, ..., <tin>
process.<id>.receiveomission.finish = <tf0>, <tf1>, ..., <tfm>
```

O parâmetro de configuração da política é:

```
process.<id>.receiveomission.dropmodel = <política>
```

A `<política>` informa o nome da classe Java que define o comportamento de descarte. Pode ainda ser definida a mesma política para todos os processos, bastando suprimir o `<id>` do parâmetro de configuração. Neste caso, todos os processos que não possuam parâmetros específicos utilizarão o parâmetro sem `<id>`.

Para adicionar a camada `ReceiveOmissionLayer`, basta substituir a linha 19 do exemplo da Figura 5.2 pela linha de comando `process.addLayer(new ReceiveOmissionLayer(process))`.

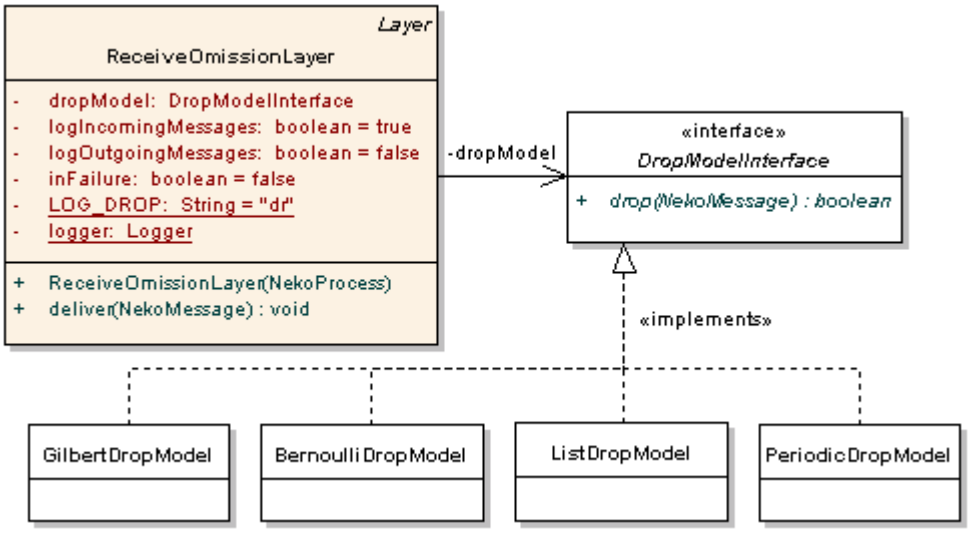


Figura 5.3: Diagrama da classe `lse.neko.layers.ReceiveOmissionLayer`

Da mesma forma que na classe `SendOmissionLayer`, para cada descarte é gerada uma linha registrando sua ocorrência no arquivo de *log* da simulação. A diferença em relação ao *log* gerado pela omissão no envio (Tabela 5.1) é o identificador da classe que gerou o evento, que neste caso é `ReceiveOmissionLayer` e o valor da coluna 5, que neste caso é "*dr*", indicando que se trata de um descarte no recebimento. As considerações referentes ao registro de mensagens no arquivo de *log* são os mesmos discutidos na classe de omissão no envio (seção 5.1.1), isto é, podem ocorrer inconsistências no arquivo de *log*, com mensagens registradas como recebidas e posteriormente registradas como descartadas. Isto pode ser evitado com o bloqueio do *log* na camada `NekoProcess`, deixando a cargo da camada de omissão no recebimento o registro de mensagens recebidas.

5.1.3 Omissão geral

Como discutido anteriormente, sistemas que apresentam defeitos de omissão geral (*general omission failures*) incluem defeitos de envio e de recebimento aleatoriamente.

Este comportamento pode ser simulado facilmente no Neko fazendo uso das classes `SendOmissionLayer` e `ReceiveOmissionLayer` desenvolvidas neste trabalho e aproveitando a estrutura de camadas do *framework*, que facilita a adição de novas funcionalidades. Para tanto, basta adicionar, ao invés de uma ou outra camada de descarte, as duas, consecutivamente. Desta forma, uma camada será responsável por interceptar e descartar mensagens no envio, através do método `send()` e a outra fica responsável por descartar mensagens no recebimento, através do método `deliver()`. A adição destas camadas segue o exemplo da Figura 5.4. A configuração das taxas de descarte é feita da mesma forma descrita nas seções anteriores, já que cada camada funciona independentemente uma da outra.

```
    ...
1   public void init(NekoProcess process, Configurations config)
2      throws Exception
3   {
4     process.addLayer(new ReceiveOmissionLayer(process));
5     process.addLayer(new SendOmissionLayer(process));
6     process.addLayer(new Lamport(process));
7   }
8 }
```

Figura 5.4: Arquivo de inicialização para simular defeitos de omissão geral

5.2 Defeitos de Colapso

A proposta para simulação de defeitos de colapso, descrita na seção 4.2 sugere a inclusão de mecanismos para interceptar as mensagens e sinalizar o início e o fim do colapso. Visando atender esta especificação, foi implementada uma classe Java abstrata chamada `CrashLayer` que faz a interceptação das mensagens enviadas e recebidas e as descarta durante o período de defeito. Além disso, foram definidos dois métodos abstratos nomeados `crash()` e `recover()`. O primeiro é utilizado para iniciar o defeito e o segundo para finalizá-lo. Estes métodos não possuem implementação na `CrashLayer`, visto que o comportamento do colapso depende do modelo desejado. Cada tipo de comportamento foi modelado em outra classe que estende a classe `CrashLayer`, de acordo com a Figura 5.5. Por esta razão, a `CrashLayer` não deve ser adicionada diretamente na pilha de camadas do processo.

Cada classe de defeito implementa seus próprios métodos `crash()` e `recover()`. Entretanto, estes métodos fazem uso de algumas funcionalidades incluídas no Neko. Estas funcionalidades incluem mecanismos para suspender e reiniciar a aplicação, tanto no ambiente simulado quanto no emulado.

Uma aplicação simulada é instanciada a partir da classe `lse.neko.sim.nekosim.SimThread` e uma aplicação emulada instancia um objeto da classe `lse.neko.comm.NekoCommThread`. Estas duas classes implementam a interface `lse.neko.NekoThreadInterface`. Assim, foram definidos alguns métodos na interface `NekoThreadInterface` que devem ser implementados nessas outras duas classes para efetuar a suspensão e o retorno da execução da aplicação. Estes novos métodos são utilizados pelas classes de colapso para efetuar o início e fim do colapso. A Figura 5.6 contém as alterações realizadas nas classes do Neko para dar suporte ao novo modelo de defeitos de colapso. Os seguintes métodos foram adicionados:

- `suspendThread()`: efetua o bloqueio da aplicação, isto é, suspende a sua execução;
- `resumeThread()`: faz com que a aplicação (previamente suspendida) volte à sua execução a partir do estado em que parou.
- `stopThread()`: finaliza a execução da aplicação. Neste caso não é possível reiniciar a aplicação a partir do estado anterior à chamada do método;
- `restartThread()`: reinicia a aplicação (previamente interrompida) a partir do estado inicial.

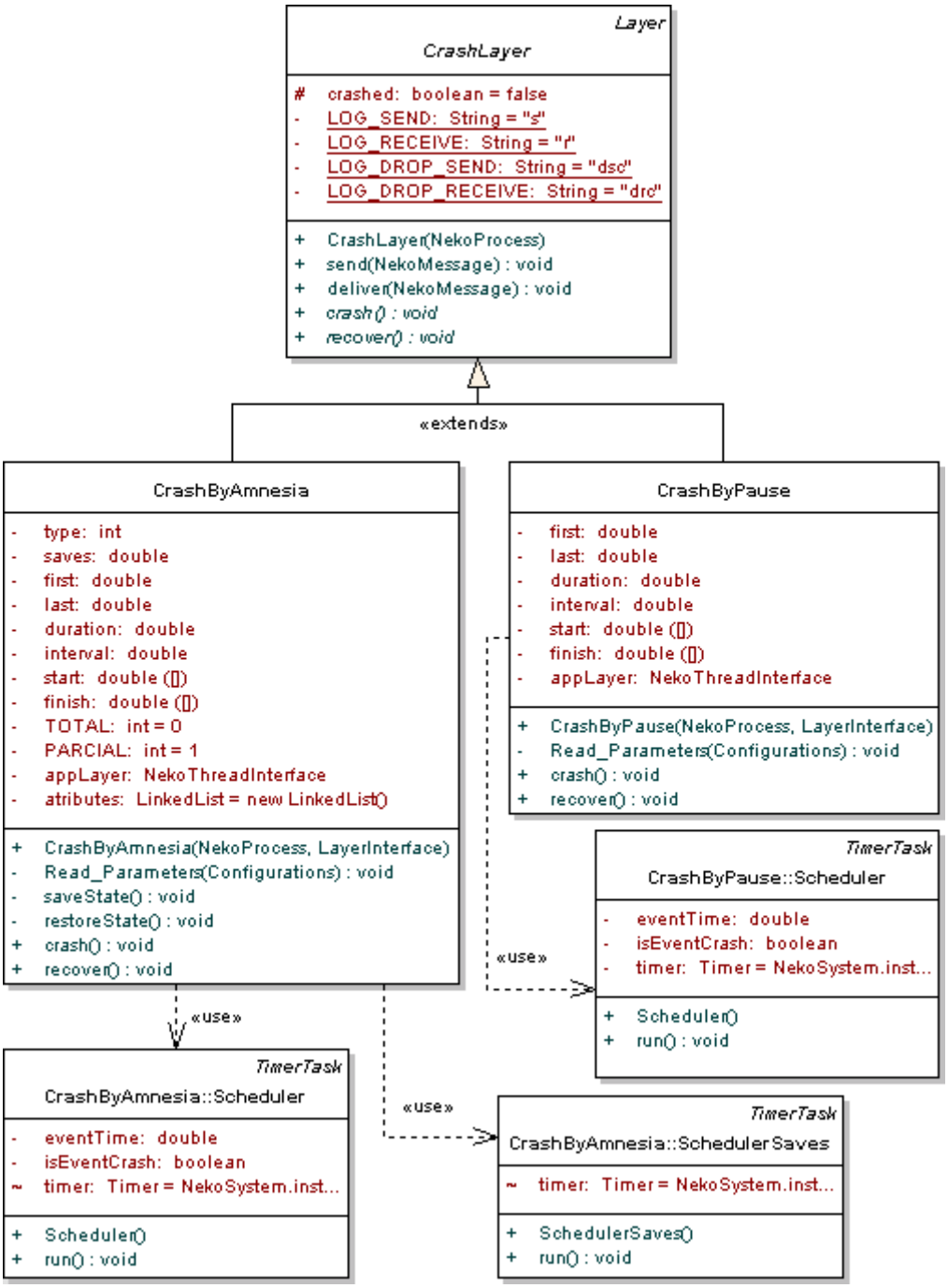


Figura 5.5: Diagrama das classes de defeitos por colapso

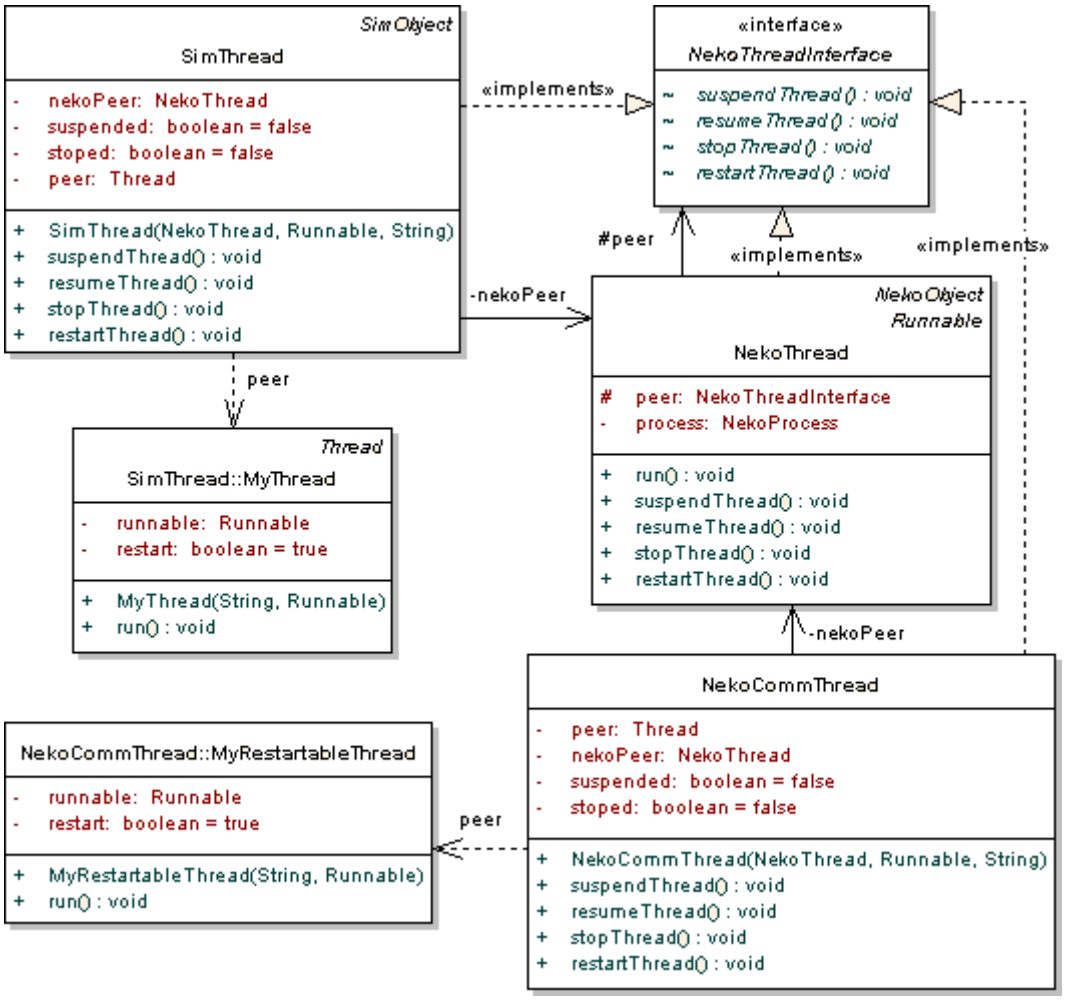


Figura 5.6: Alterações no Neko para simular defeitos de colapso

Os métodos `suspendThread()` e `resumeThread()` são utilizados para gerar colapsos por pausa e parada, e os métodos `stopThread()` e `restartThread()` são utilizados para simular colapsos com amnésia parcial e total. A diferença é que nestes dois últimos métodos, a aplicação é realmente finalizada e volta a ser executada a partir do início.

O comportamento e utilização destes métodos para cada tipo de defeito é exposto de forma mais detalhada nas próximas seções.

5.2.1 Colapso por pausa

A solução para simulação de colapsos por pausa utiliza os métodos disponibilizados nas classes `CrashLayer` e `NekoThread` descritos na seção anterior. Para tanto, definiu-se uma classe chamada `CrashByPause`, que estende a classe `CrashLayer` e implementa os métodos `crash()` e `recover()`.

O método `crash()` solicita o bloqueio da aplicação sendo simulada ou emulada através de `suspendThread()` e o método `recover()` utiliza `resumeThread()` para permitir o retorno da execução da aplicação. Não há qualquer alteração no estado da aplicação, a qual continua sua execução a partir do ponto em que estava antes do bloqueio.

A configuração da ocorrência deste tipo de defeito necessita de alguns parâmetros no arquivo de configuração. O parâmetro `first` define o tempo em que ocorrerá o primeiro colapso, o parâmetro `duration` informa o tempo de duração do colapso e o

parâmetro `interval` define o intervalo entre um colapso e outro. O parâmetro `last` informa quando deverão cessar os defeitos, podendo ser omitido. Neste caso, os defeitos continuarão ocorrendo até o final do experimento.

Como exemplo, para uma simulação com `first = 200`, `interval = 100` e `duration = 50` serão ocasionados defeitos nos tempos 200, 350, 500, 650, ... e assim por diante até o fim da simulação.

Estes parâmetros devem estar no formato a seguir:

```
process.<id>.crash.pause.first = <tempo_inicial>
process.<id>.crash.pause.duration = <tempo_de_duração>
process.<id>.crash.pause.interval = <tempo_de_intervalo>
process.<id>.crash.pause.last = <tempo_final>
```

O colapso por pausa pode ainda ser configurado para ocorrer em tempos pré-determinados utilizando-se os parâmetros:

```
process.<id>.crash.pause.start = <ti0>,<ti1>, ..., <tin>
process.<id>.crash.pause.finish = <tf0>, <tf1>, ..., <tfm>
```

com $n\text{-}1 \leq m \leq n$.

Cada tempo inicial (ti_i) é mapeado em um tempo final (tf_i). A diferença (tf_i-ti_i) define a duração do defeito. É possível ainda existir um último tempo ti_n que não possua seu respectivo tf_n. Neste caso, o defeito se tornará perpétuo e o processo não mais se recuperará até o final do experimento.

O valor de `<id>` pode ser suprimido em qualquer parâmetro descrito acima. Neste caso, o parâmetro poderá ser utilizado por qualquer outro processo que possua a camada de defeito de colapso e não possua parâmetro específico, isto é, com seu `<id>`. Como exemplo, se o processo com identificador 1 (*p1*) necessita do parâmetro `start` e não encontra o parâmetro `process.1.crash.pause.start`, ele irá utilizar `process.crash.pause.start`, caso este exista. Se nenhum destes parâmetros for encontrado, será gerado um erro informado a falta do parâmetro em questão.

Estas duas formas de configuração (em intervalos regulares e em tempos pré-determinados) não foram implementadas para serem utilizadas em conjunto e a segunda forma será utilizada somente na ausência da primeira.

O escalonamento das chamadas `crash()` e `recover()` é feito com o auxílio da classe `lse.neko.util.Timer`. Esta classe possui um método `scheduler(task, delay)` que pode ser utilizado para escalonar uma tarefa (`task`) a ser executada a partir de um tempo específico (`delay`). Através deste recurso, os métodos `crash()` e `recover()` são escalonados de acordo com os tempos informados.

5.2.2 Colapso por parada

A solução para simular defeitos de colapso por parada utiliza os mesmos mecanismos desenvolvidos para a simulação de colapso por pausa. Neste caso, uma vez que o processo venha a parar, este não se recuperará mais. Assim, a classe de parada classe simplesmente faz o escalonamento do primeiro colapso utilizando a classe `lse.neko.util.Timer`, descrita na seção anterior. Todas as mensagens enviadas e recebidas serão descartadas durante o colapso e o processo não mais se recuperará. Isso significa que o método `recover()` jamais será invocado.

O único parâmetro que deve ser definido no arquivo de configuração é o tempo em que o defeito deverá ocorrer, seguindo a seguinte sintaxe:

```
process.<id>.crash.pause.start = <tempo_inicial>
```

5.2.3 Colapso por amnésia total e amnésia parcial

Como já mencionado, o colapso de processos por amnésia foi dividido em parcial e total. A classe Java desenvolvida foi nomeada `CrashByAmnesia`. A diferença entre os dois tipos de amnésia está no estado do processo que é restaurado após o retorno do defeito. Após uma amnésia total o processo volta ao estado inicial e após uma amnésia parcial o processo é recuperado a partir de um estado intermediário, salvo previamente.

O salvamento de estado do processo foi implementado utilizando-se a biblioteca de reflexão do Java (`java.lang.reflect`). Através desta biblioteca é possível ter acesso a classes, atributos e métodos de um objeto Java qualquer. A principal vantagem desta solução é que não há necessidade de se conhecer de antemão a classe Java que o usuário do Neko desenvolveu para ser simulada. Basta obter uma referência para esta classe a fim de ter acesso a seus atributos e métodos.

Uma das restrições do pacote `reflect` é que ele permite a recuperação do valor de um atributo apenas quando o mesmo é público. Para contornar esta impossibilidade, definiu-se que a classe do usuário deverá conter métodos `get` e `set` para cada atributo do qual se deseja salvar o estado durante o colapso. Os métodos `get` são utilizados para obter os valores dos atributos e os métodos `set` para alterar esses valores de acordo com àqueles previamente salvos. Neste caso, por exemplo, se o algoritmo possui um atributo `mensagensEnviadas` do tipo `long` cujo estado deve ser salvo durante o defeito, é preciso definir um método `public long getMensagensEnviadas()` e um outro, também público, `setMensagensEnviadas(long valor)`.

Para simular o colapso com amnésia total são realizados os seguintes passos:

1. o estado inicial do algoritmo sendo simulado é salvo;
2. na ocorrência do defeito, a aplicação é bloqueada e o estado inicial é restaurado;
3. ao final do defeito, a aplicação é reiniciada a partir do estado inicial, já restaurado.

A simulação de amnésia parcial foi implementada de modo a simular o salvamento do estado do processo a partir de intervalos. A cada intervalo de tempo o estado do processo é salvo e, na ocorrência do defeito, o ultimo estado salvo é restaurado.

Os parâmetros utilizados para configurar o colapso com amnésia (parcial ou total) são semelhantes ao modelo de configuração de colapso por pausa e incluem:

```
process.<id>.crash.amnesia.type = <total | parcial>
process.<id>.crash.amnesia.first = <tempo_inicial>
process.<id>.crash.amnesia.duration = <tempo_de_duração>
process.<id>.crash.amnesia.interval = <tempo_de_intervalo>
process.<id>.crash.amnesia.last = <tempo_final>
```

A distinção entre amnésia total e parcial é feita pelo parâmetro `type`.

Da mesma forma que no colapso por pausa, o colapso com amnésia pode ainda ser configurado para ocorrer em tempos pré-determinados utilizando-se os parâmetros:

```
process.<id>.crash.amnesia.start = <ti0>,<ti1>, ..., <tin>
```

`process.<id>.crash.amnesia.finish = <tf0>, <tf1>, ..., <tfm>`

Se a amnésia é `parcial`, deve ser definido ainda o intervalo de salvamento do estado do processo, utilizando-se o seguinte parâmetro:

`process.<id>.crash.amnesia.saves = <intervalo_salvamento>`

Em ambos os casos, parcial ou total, o estado inicial é salvo no início do experimento. Este estado inicial é utilizado sempre que ocorrer uma amnésia total ou quando ocorrer uma amnésia parcial entre o início da simulação e o segundo salvamento de estado, isto é, antes do segundo salvamento.

5.3 Defeitos de Rede e Particionamento

Visando atender as especificações descritas na modelagem para os defeitos de rede e particionamento (seção 4.3), foram desenvolvidas as classes Java ilustradas na Figura 5.7.

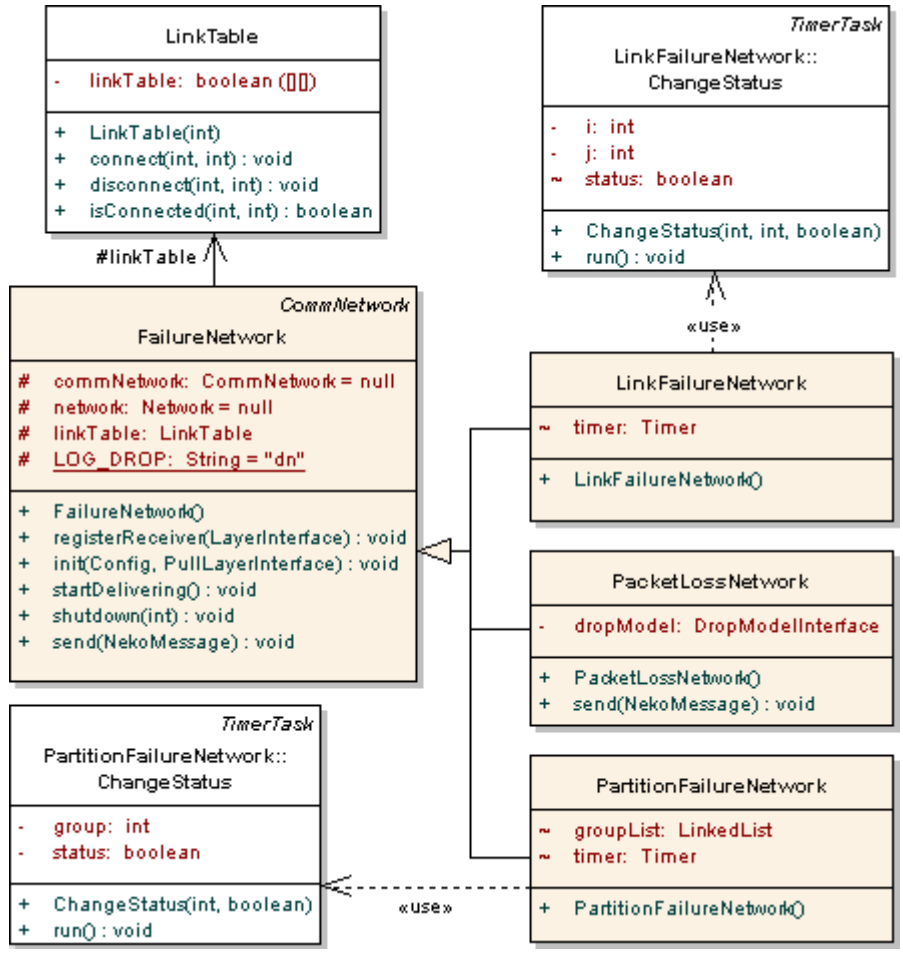


Figura 5.7: Diagrama das redes para simulação de defeitos de rede e particionamento

A classe `FailureNetwork` mantém uma matriz de conexão entre os processos da simulação, já representada no capítulo anterior (Figura 4.6 da seção 4.3).

Uma ligação entre processos pode estar em estado UP (quando a conexão está habilitada) ou DOWN (quando a conexão está desfeita). Antes de transmitir qualquer mensagem, a rede `FailureNetwork` verifica o estado da conexão entre o processo

emissor e o(s) processo(s) destino. Se o valor é DOWN a mensagem é descartada e o descarte é registrado no arquivo de *log*. Se o estado é UP, a mensagem é transmitida normalmente.

Para alterar o estado do link entre dois processos deve ser utilizado o método `SetLinkStatus(i,j,status)`, onde `i` é o identificador do processo de origem, `j` é o identificador do processo destino e `status` é o novo valor do enlace (`true` para UP ou `false` para DOWN). Como as conexões são bidirecionais, toda vez que o estado é alterado, a matriz deve ser atualizada tanto na célula `i,j` quanto na célula `j,i`, embora isto seja feito automaticamente.

A classe `FailureNetwork`, por si só, não possui meios pelos quais o usuário possa alterar o estado da conexão entre processos em tempo de execução. Para tanto, foram desenvolvidas duas outras classes que estendem a classe `FailureNetwork`, chamadas `LinkFailureNetwork` e `PartitionFailureNetwork`, descritas a seguir.

5.3.1 Defeitos de rede

Os defeitos de rede foram divididos em defeitos de conexão entre processos e defeitos ocasionados pelo descarte de mensagens na rede. As classes que modelam estes defeitos são `LinkFailureNetwork` e `PacketLossNetwork`, respectivamente.

Uma rede `LinkFailureNetwork` possui suporte a alterações no estado dos enlaces entre um ou mais processos, com já descrito na seção anterior. A configuração é feita por meio do parâmetro:

`network.link.<id1>.<id2>.[up | down] = <t1,t2,…,tn>`

O `<id1>` indica o identificador do processo origem, o `<id2>` indica o identificador do processo destino. Para determinar a mudança do estado no mesmo *link* mais de uma vez, basta separar os tempos por vírgula. No exemplo da Figura 5.8, o primeiro parâmetro indica ao Neko que a rede `LinkFailureNetowork` será utilizada. Em seguida, é preciso indicar qual rede já disponível no Neko será responsável pelo envio das mensagens. Os parâmetros `lambda` e `multcast` são específicos desta rede. Esta solução foi utilizada para evitar que todas as redes já desenvolvidas precisassem ser alteradas. Assim, a rede `LinkFailureNetwork` somente faz a análise de descarte, deixando o envio para a rede informada no parâmetro `network.net`.

De acordo com o exemplo, o enlace entre o processo *p0* e o processo *p1* é desfeito no tempo 10, refeito no tempo 20, desfeito no tempo 30 e refeito no tempo 50. Durante o período em que o *link* está no estado DOWN, todas as mensagens transmitidas entre *p0* e *p1* são descartadas. Nos períodos em que o *link* está no estado UP as mensagens serão enviadas utilizando a rede `lse.neko.networks.sim.MetricNetwork`.

```
network = LinkFailureNetwork
network.net = lse.neko.networks.sim.MetricNetwork
network.lambda = 0.061
network.multicast = false

network.link.1.0.down = 10.0,30.0
network.link.1.0.up   = 20.0,50.0
```

Figura 5.8: Configuração para alteração no estado dos links

Outra forma de gerar descartes utiliza a classe `PacketLossNetwork`. Os métodos de descarte disponíveis são os mesmos descritos e utilizados nos defeitos de omissão. A

configuração do modelo de descarte é semelhante ao descrito na omissão. Para informar qual será o modelo empregado e quais as informações que o modelo escolhido necessita deve-se informar os parâmetros:

```
network.dropmodel = <política_de_descarte>
network.dropmodel.params = <parâmetros>
```

Os <parâmetros> variam de acordo com o modelo e estão descritos na seção 5.4.

Uma diferença do descarte efetuado pela rede daqueles gerados pela omissão é que não existe distinção entre os processos que terão mensagens descartadas. Na rede, todas as mensagens têm a mesma probabilidade de descarte. Obviamente que os processos que enviam mais mensagens possuem maior probabilidade de terem suas mensagens descartadas. Todas as mensagens descartadas são registradas no arquivo de *log* da simulação com identificador dn (*Droped by Network*).

5.3.2 Defeitos de particionamento

A classe Java PartitionFailureNetwork foi desenvolvida com o propósito de facilitar a geração de defeitos de particionamento. É possível verificar que utilizando a classe LinkFailureNetwork, descrita acima, pode-se obter o particionamento de um processo na rede isolando-o dos demais. Entretanto, como cada processo mantém um enlace direto de comunicação com todos os outros, em uma simulação com um grande número de processos seria necessário escrever diversas linhas de parâmetro para desativar todos os enlaces. O mesmo problema seria desfazer o particionamento, isto é, ativar todos os enlaces novamente. Neste sentido, foi desenvolvido um mecanismo que permite a definição de grupos de processos. Uma vez definido um grupo, é possível iniciar e finalizar o particionamento do mesmo em relação aos demais processos na rede que não estejam incluídos no grupo em questão. A matriz de ligação entre processos e os métodos para ativação e desativação dos enlaces são herdados da classe FailureNetwork.

Para definir os grupos deve-se primeiramente utilizar os parâmetros:

```
network.groups = <n>
network.groups.<g0> = <lista_de_processos>
...
network.groups.<gn-1> = <lista_de_processos>
```

O valor de <n> estabelece o número de grupos que serão formados. Em seguida, deve ser feita a definição dos grupos, onde $<g_i>$ é o identificador do grupo, sendo $0 \leq i \leq n-1$. A <lista_de_processos> contém os identificadores dos processos contidos no grupo, separados por vírgula (“,”).

A Figura 5.9 apresenta um exemplo de configuração de grupos. O primeiro parâmetro indica ao Neko que a rede PartitionFailureNetowork será utilizada. Em seguida, é preciso indicar qual rede já disponível no Neko será responsável pelo envio das mensagens (parâmetro network.net). Da mesma forma como foi descrito na seção anterior, esta solução foi utilizada para evitar que todas as redes já desenvolvidas precisassem ser alteradas. O parâmetro network.groups informa que serão formados dois grupos, definidos em seqüência. O grupo 0 contém os processos *p0*, *p1* e *p2* e o grupo 1 contém os processos *p3* e *p4* (esta configuração equivale a Figura 4.6 da seção 4.3).

```
network = PartitionFailureNetwork
network.net = lse.neko.networks.sim.MetricNetwork
network.lambda = 0.061

network.groups = 2
network.group.0 = 0,1,2
network.group.1 = 3,4

network.group.0.partition.start = 100.0, 500.0
network.group.0.partition.finish = 150.0, 550.0
```

Figura 5.9: Exemplo de configuração de defeitos de particionamento

Da mesma forma como foi implementada a quebra de enlaces da seção anterior, é preciso definir o início e fim do particionamento. Esta configuração é feita por grupos. No exemplo, os membros do grupo 0 são particionados dos demais processos no tempo `100.0`, voltam a se comunicar no tempo `150.0` e são novamente isolados no tempo `500.0`. No instante `550.0` o grupo 0 é reintegrado aos demais processos e não sofre mais particionamento até o final da simulação.

A configuração dos grupos fica a cargo do usuário. Neste sentido, podem ser feitas algumas considerações sobre tal configuração:

1. não é necessário incluir cada processo em um grupo. Da mesma forma que um mesmo processo pode estar contido em mais de um grupo, de acordo com a necessidade da simulação;
2. a construção de um grupo não implica na existência de uma configuração de início e fim do particionamento para este grupo. Neste caso, o próprio grupo pode ser retirado da configuração. No exemplo dado, o grupo 1 não interfere na simulação e poderia ser retirado.

5.4 Modelos de Descarte

De acordo com os modelos de descarte descritos na seção 4.5, foram criadas as classes Java da Figura 5.10. Cada modelo necessita de parâmetros específicos, que são repassados através do arquivo de configuração. Sabendo que estes modelos podem ser utilizados tanto nos defeitos de omissão quanto no defeito de rede gerado pela `PacketLossNetwork`, os parâmetros devem ser lidos do arquivo de configuração de forma apropriada.

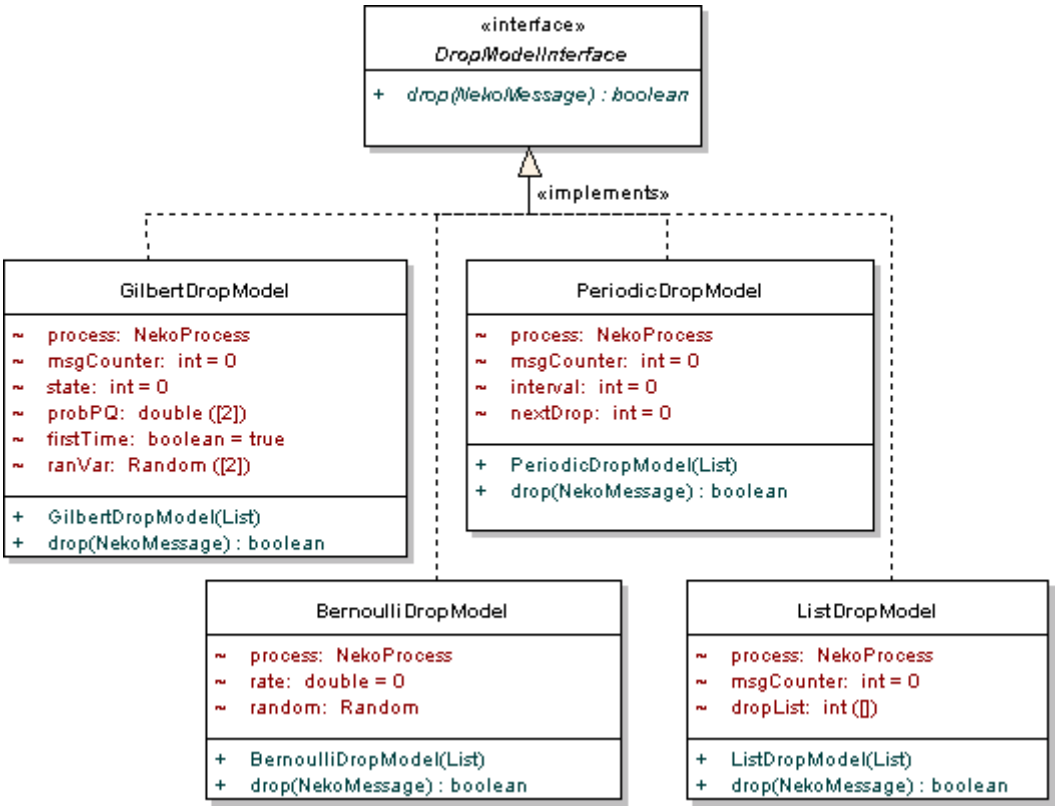


Figura 5.10: Políticas de descarte de mensagens

5.4.1 Modelo de Gilbert

O modelo de Gilbert (`GilbertDropModel`) necessita de dois parâmetros: <p> e <q>, onde <p> é a probabilidade de mudança para o estado de descarte e <q> é a probabilidade de mudança para o estado de não-descarte. É válido lembrar que <p> e <q> são valores decimais no intervalo [0,1]. A sintaxe dos parâmetros é a seguinte:

a) omissão

```
process.<id>.[sendomission|receiveomission].dropmodel.params =
      <p>,<q>
```

b) rede

```
network.dropmodel.params = <p>,<q>
```

5.4.2 Modelo de Bernoulli

O modelo de Bernoulli (`BernoulliDropModel`) necessita de um único parâmetro, que representa a taxa <r> de descarte a ser aplicada, devendo ser um valor decimal entre 0 e 1. A configuração é:

a) omissão

```
process.<id>.[sendomission|receiveomission].dropmodel.params =
<r>
```

b) rede

```
network.dropmodel.params = <r>
```

5.4.3 Descarte em Intervalos Regulares

O descarte periódico (`PeriodicDropModel`) necessita apenas do intervalo `<i>` entre descartes. Este valor é referente ao número de mensagens e não ao tempo de simulação. Os parâmetros são:

a) omissão

```
process.<id>.[sendomission|receiveomission].dropmodel.params = <i>
```

b) rede

```
network.dropmodel.params = <i>
```

5.4.4 Lista de Descarte

O último modelo de descarte (`ListDropModel`) utiliza uma lista com identificadores das mensagens que serão descartadas, separados por vírgula (“,”). Neste caso, devem ser utilizados os seguintes parâmetros:

a) omissão

```
process.<id>.[sendomission|receiveomission].dropmodel.params = <m1>, <m2>,..., <mn>
```

b) rede

```
network.dropmodel.params = <m1>, <m2>,..., <mn>
```

6 ESTUDOS DE CASO E AVALIAÇÃO

Este capítulo dedica-se aos estudos de caso empregados na utilização das soluções de simulação de defeitos descritas nos capítulos 4 e 5. Primeiramente são apresentadas algumas aplicações que foram desenvolvidas para serem utilizadas em conjunto com as classes de defeitos. A aplicação da seção 6.1 é utilizada na camada mais superior da pilha dos processos, sendo responsável pelo envio e recebimento de mensagens de aplicação. Esta aplicação foi empregada nos experimentos com defeitos de omissão e colapso.

Outras duas camadas foram implementadas para serem usadas em conjunto com a aplicação da seção 6.1 nos experimentos com os defeitos de colapso. Estas incluem um algoritmo de *checkpointing* e um detector de defeitos. O algoritmo de *checkpointing* foi escolhido por ser um trabalho desenvolvido pelo grupo de Tolerância a Falhas e por considerar defeitos de colapso na sua implementação. O desenvolvimento do detector foi necessário porque o algoritmo de *checkpointing* necessita de um mecanismo de detecção de defeitos para ativar a recuperação da aplicação. Embora existam alguns detectores de defeitos já incluídos no Neko, decidiu-se desenvolver este como um exemplo adicional de implementação.

A aplicação sintética (seção 6.1) é também utilizada para demonstrar o funcionamento dos defeitos de enlace. Já os experimentos com defeitos de particionamento de rede utilizaram a aplicação disponível no pacote de exemplos do Neko (`lse.neko.examples.basic`), que simula o modelo de comunicação do algoritmo de consenso de Lamport (1978).

A partir da seção 6.4 são apresentados os resultados obtidos com as execuções das aplicações em conjunto com ambiente de simulação de defeitos desenvolvidos no trabalho. Todos os testes realizados, seus resultados e arquivos de configuração estão disponíveis no CD-ROM do Apêndice B.

6.1 Aplicação Sintética

Visando avaliar e exemplificar o funcionamento das soluções implementadas no decorrer do trabalho foi utilizado uma aplicação sintética, descrita em Buligon (2005). A aplicação consiste de um servidor e diversos clientes (Figura 6.1). Os clientes enviam um número *x* fixo de mensagens ao servidor e, de acordo com a configuração, aguardam ou não uma mensagem de reconhecimento (ACK - *Acknowledgment*), sendo um ACK para cada mensagem enviada. O servidor por sua vez, ao receber as mensagens dos clientes, simula um tempo de processamento e, se necessário, responde ao cliente com um ACK. O número de clientes é sempre *n-1*, onde *n* representa o número de processos participantes. O processo com identificador *0* (*p0*) assume o papel de servidor.

Os processos clientes possuem os seguintes parâmetros de configuração:

- *packets*: informa o número de mensagens que serão enviadas;

- *tcpu*: tempo médio (em milissegundos) dedicados a gerar carga de CPU entre o envio de cada mensagem;
- *twait*: tempo de espera após o envio da mensagem, em milissegundos;
- *waitforack*: opção para informar se haverá espera de mensagens de reconhecimento para as mensagens enviadas.
- *timeout*: período máximo de espera (em milissegundos) utilizado nos casos em que ocorre descarte de mensagens, seja por omissão ou colapso. Sem este parâmetro, o processo pode ficar bloqueado nos casos de espera pelo ACK.

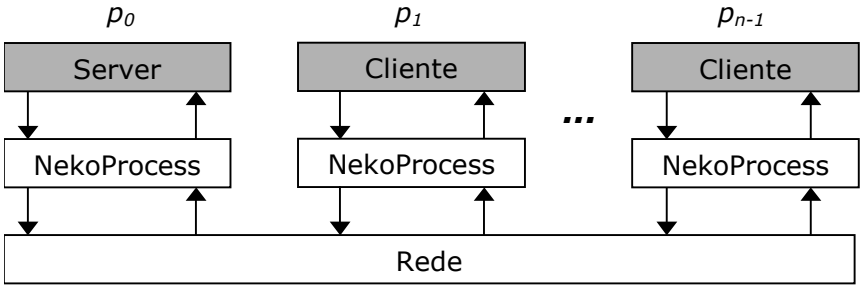


Figura 6.1: Arquitetura da aplicação sintética utilizada nas simulações

O processo servidor também faz uso dos parâmetros *tcpu* e *waitforack*. O primeiro é utilizado da mesma forma que nos clientes. O último determina se as mensagens de reconhecimento deverão ser enviadas aos clientes.

Através dos parâmetros disponibilizados é possível modificar o comportamento da aplicação, podendo por exemplo, simular um maior uso da CPU ou da rede. Assim como em Buligon (2005) foram utilizadas 5 configurações diferentes nos testes: NET, BAL, CPU, LOW e HIGH (Tabela 6.1).

Tabela 6.1: Parâmetros utilizados em cada tipo de aplicação

Parâmetros	Aplicações				
	NET	BAL	CPU	LOW	HIGH
packets	60000	40000	10000	30000	600000
tcpu	0	0	3	0	0
twait	0	1	0	0,003	0
waitforack	true	true	true	true	false

A aplicação NET simula o uso intenso de rede e pouco uso de CPU. A aplicação CPU simula maior tempo de processamento e pouca utilização da rede. A aplicação BAL faz uso balanceado entre rede e processamento. A aplicação LOW não executa qualquer tipo de processamento e restringe-se ao envio de mensagens seguido por um período de espera. A última aplicação, HIGH, faz uso extremo da rede, enviando mensagens sem aguardar as mensagens de reconhecimento e sem utilizar tempo para processamento.

6.2 Algoritmo de *Checkpointing*

Um outro algoritmo escolhido como estudo de caso é o algoritmo de *checkpointing* proposto por Cechin (2002). Trata-se de um algoritmo coordenado onde periodicamente são estabelecidos pontos de recuperação. No caso de ocorrência de defeitos, estes pontos serão utilizados para reiniciar a aplicação a partir do último estado consistente. Da forma como foi projetado, o algoritmo não exige que a aplicação monitorada seja

interrompida durante o estabelecimento dos pontos de recuperação, exceto durante o salvamento do estado em memória estável.

O estabelecimento de uma nova linha de recuperação utiliza um protocolo de duas fases (Figura 6.2). Um processo eleito coordenador estabelece um ponto de recuperação local e envia uma mensagem do tipo `REQ` aos demais processos propondo o estabelecimento de uma nova linha de recuperação. Uma vez recebida a mensagem do coordenador, todos os processos estabelecem seus pontos de recuperação e enviam uma mensagem do tipo `AREQ` ao coordenador, confirmando o estabelecimento do *checkpoint*.

Cada processo mantém a última linha de recuperação, chamada de *PrevCP*, para o caso de ocorrer um defeito durante o estabelecimento do novo ponto de recuperação (*CurrCP*). O coordenador, ao receber a mensagem de confirmação dos demais processos, envia uma nova mensagem do tipo `CMT` informando que a coleta de lixo pode ser realizada. Neste ponto, *PrevCP* é substituído pela nova linha de recuperação. Os processos respondem novamente ao coordenador através de uma mensagem do tipo `ACMT` informando que a coleta de lixo foi realizada com sucesso. Para finalizar, o coordenador encerra a operação de estabelecimento da linha de recuperação e está pronto para iniciar uma nova.

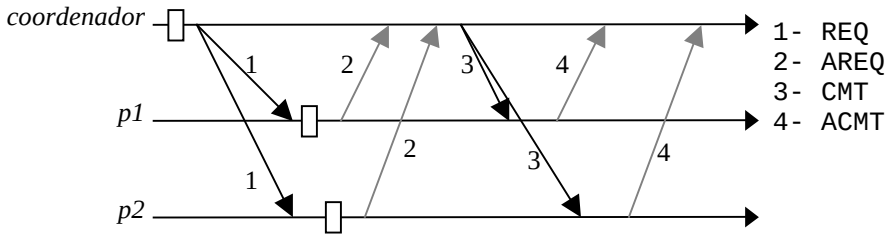


Figura 6.2: Protocolo para estabelecimento de uma nova linha de recuperação

Cada processo pode ainda estabelecer pontos de recuperação locais antes mesmo de receber uma mensagem `REQ` do coordenador. Isto pode acontecer porque o algoritmo de *checkpointing* não exige que a aplicação pare sua execução durante o processo de estabelecimento de uma nova linha de recuperação. Isto significa que mensagens da aplicação continuarão sendo enviadas em conjunto com as mensagens de controle do algoritmo. Uma vez que cada mensagem carrega consigo o número do último ponto de recuperação armazenado pelo emissor, toda vez que uma nova mensagem for recebida, este número será comparado com o identificador do último ponto de recuperação do receptor. Caso a mensagem recebida possua um identificador maior, um novo ponto de recuperação deve ser realizado antes do recebimento da mensagem. Isto deve ser feito para evitar mensagens perdidas, visto que não há garantia de ordenação das mensagens nos canais de comunicação (CECHIN, 2002). Nestes casos, quando o processo receber a mensagem `REQ` do coordenador, não será necessário estabelecer um novo ponto de recuperação. O processo apenas responderá com um `AREQ`, pois o ponto solicitado já foi estabelecido.

Para testar a validade do algoritmo proposto, foi necessário incluir um detector de defeitos na simulação (seção 6.3). Assim, uma vez detectado o defeito, o algoritmo de *checkpointing* deverá realizar as operações de retorno, nas quais todos os processos deverão retornar ao último ponto de recuperação consistente. Esta etapa consiste em determinar o identificador que representa o último ponto de recuperação presente em todos os processos e reiniciar a aplicação a partir deste ponto.

6.3 Detector de Defeitos

Para ser utilizado em conjunto com o algoritmo de *checkpointing*, proposto por Cechin (2002) e implementado por Buligon (2005), foi modelado no Neko o detector de defeitos baseado no modelo de monitoração apresentado por Fetzer, Raynal e Tronel (2001). Este detector permite que um processo monitore outros processos para detectar a ocorrência de colapso (*crash*). A principal característica deste protocolo é que ele utiliza as próprias mensagens de aplicação para realizar o monitoramento, deixando as mensagens de controle, isto é, aquelas geradas pelo próprio detector, apenas para quando as mensagens de aplicação não são trocadas entre o processo monitor e o processo monitorado. Devido a esta característica, este detector é conhecido como *lazy*.

Para cada processo da aplicação existe uma instância do detector associada (Figura 6.3). Durante a execução da aplicação, o módulo detector intercepta e armazena em uma lista todas as mensagens trocadas entre os processos. No lado do processo receptor, o módulo detector intercepta as mensagens que estão sendo recebidas e, para cada mensagem, responde ao emissor com uma mensagem de reconhecimento (ACK). Quando a mensagem de reconhecimento é recebida, o detector do lado emissor retira a mensagem correspondente da lista de mensagens enviadas. Além dessa lista de mensagens enviadas, cada processo mantém uma lista de processos com os quais se comunica. Toda vez que uma mensagem é recebida, esta lista é atualizada com o tempo de recebimento da mensagem.

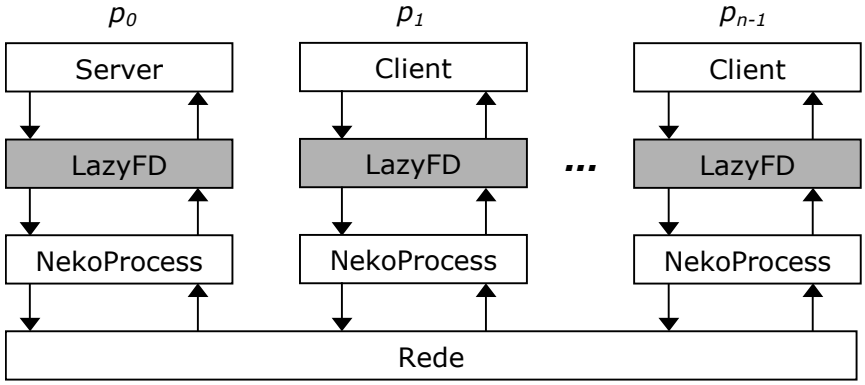


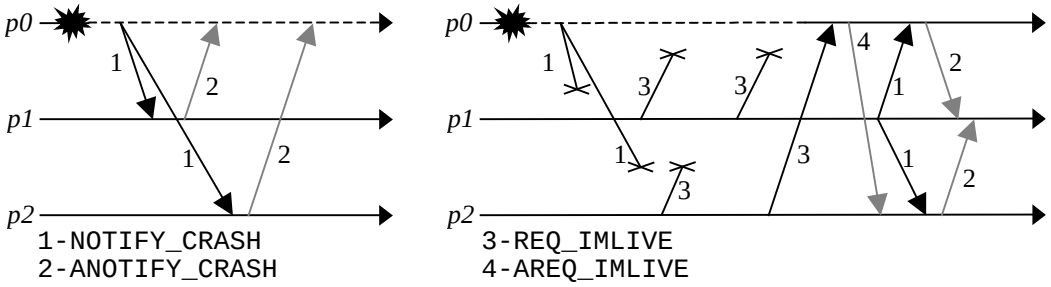
Figura 6.3: Arquitetura do detector de defeitos utilizado nas simulações

Assim como em Buligon (2005), o funcionamento do detector baseia-se em verificações periódicas realizadas em duas etapas. Na primeira etapa (Figura 6.4-a), o detector verifica o estado do processo local ao qual está associado. Caso seja detectado o defeito, o detector notifica os demais membros do grupo enviando uma mensagem do tipo `NOTIFY_CRASH` e aguarda até que todos respondam com uma mensagem do tipo `ANOTIFY_CRASH`, certificando que todos os processos foram notificados com sucesso. Em seguida o processo passa para o estado de recuperação. Os demais processos, ao receberem a mensagem de notificação do defeito, encerram o seu processamento e também passam para o estado de recuperação.

Na Figura 6.4 as linhas representam o estado dos processos e as setas ilustram as mensagens trocadas entre os módulos detectores. A linha tracejada na etapa 1 significa que o processo *p0* entrou em colapso, mas o detector local não. Já na etapa 2, houve uma falha de comunicação e tanto o processo *p0* quanto o detector local estão isolados.

A segunda etapa (Figura 6.4-b) visa detectar defeitos de comunicação entre os demais membros do grupo. Para isto deverá ser feita a verificação da lista de mensagens enviadas para os demais processos. Se a lista de mensagens enviadas estiver vazia,

significa que não existem mensagens sem reconhecimento. Neste caso, o detector interpreta que os processos estão funcionando corretamente, mas envia uma mensagem de controle do tipo `REQ_IMLIVE` para se certificar de sua decisão. Se existem mensagens enviadas e que ainda não foram reconhecidas, é feito o cálculo para verificar se ocorreu o *timeout* de resposta, subtraindo-se o valor do relógio local do tempo de envio da primeira mensagem não reconhecida e comparando com o tempo máximo de espera do ACK. Se este tempo for superior ao tempo máximo, o processo é dito suspeito.



(a) etapa 1 (b) etapa 2

Figura 6.4: Etapas para a detecção de defeitos

Em caso de suspeita, o processo é submetido a uma sondagem através do envio de mensagens do tipo `REQ_IMLIVE`. Ao receber esta mensagem, o processo suspeito responde com uma mensagem do tipo `AREQ_IMLIVE`. Isto significa que a suspeita estava incorreta. Entretanto, caso não seja recebida uma resposta após duas tentativas, o defeito é confirmado e os processos são notificados e entram em estado de recuperação. Nesta segunda etapa é possível que mais de um módulo detector detecte a ocorrência do defeito. Neste caso ocorrerão múltiplas notificações, porém não afetando o funcionamento do algoritmo.

6.4 Estudo de Caso para Defeitos de Omissão

A aplicação sintética descrita na seção 6.1 foi utilizada para exemplificar a utilização das classes de simulação de defeitos de omissão. Criou-se uma pilha de camadas em cada processo de acordo com a Figura 6.5. As camadas `SendOmissionLayer` e `ReceiveOmissionLayer` foram adicionadas a pilha dos clientes. Ao processo *p*0, foi atribuída a função de `Server` e aos demais processos a função de `Clients`.

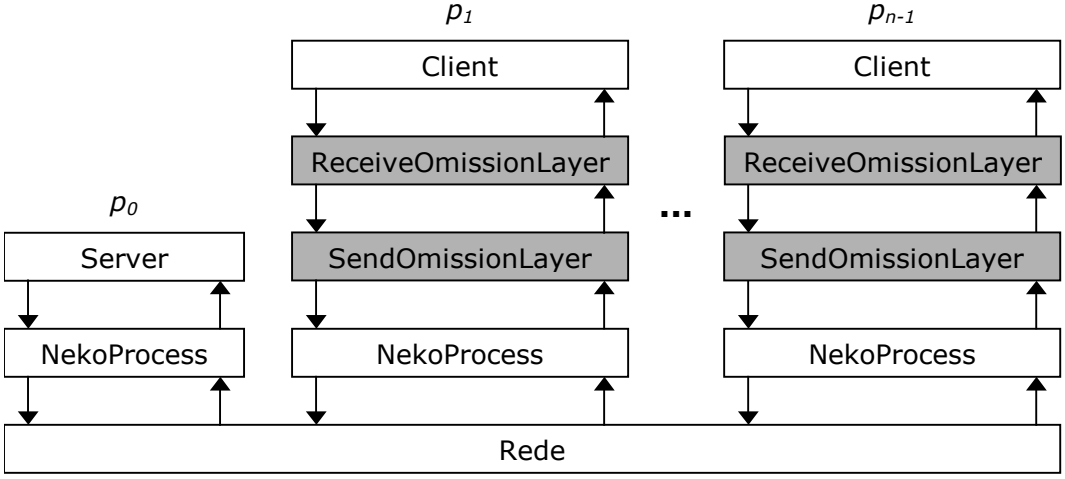


Figura 6.5: Pilha de camadas utilizada pra simular defeitos de omissão

6.4.1 Omissão no envio

No primeiro experimento apenas a classe de defeitos de omissão no envio (`SendOmissionLayer`) foi utilizada. A classe `ReceiveOmissionLayer`, embora presente, foi configurada para não gerar descartes no recebimento. Foram utilizados dois processos no experimento: um `Server` e um `Client`. O `Client` foi configurado para enviar mensagens do tipo APP e aguardar as mensagens de reconhecimento (ACKs) do `Server`. A espera pelo reconhecimento foi configurada para aguardar por um período pré-determinado (*timeout*) de 30*ms,* suficientemente grande para garantir que mensagens atrasadas fossem recebidas corretamente e não fossem incluídas como descartadas. Se a mensagem de ACK não for recebida durante este período de espera, a mensagem previamente enviada é dada como perdida, mas não é retransmitida. Este intervalo de espera poderá influenciar o tempo total de simulação quando uma mensagem for realmente descartada, visto que o processo aguardará até que o valor de *timeout* seja alcançado para enviar a próxima mensagem. Entretanto, este impacto no tempo de simulação não deve ser levado em consideração, visto que o maior interesse neste experimento está na quantidade de mensagens descartas e não no tempo de simulação.

A taxa de perda registrada na Tabela 6.2 apresenta os testes realizados utilizando o modelo de descarte de Bernoulli com taxa de descarte configurada em 10%. Estes testes foram realizados para se determinar o número de repetições necessárias para se obter resultados com um intervalo de confiança de 95%. O método utilizado no cálculo foi o descrito em Jain (1991), considerando uma margem de erro de 2,5%.

É possível perceber que quanto menor o número de mensagens, maior é a necessidade de repetições. Isto ocorre porque a variação dos valores obtidos em relação a média é proporcionalmente maior, como mostrado pelo desvio padrão.

Os dados também mostram que quanto maior o número de mensagens enviadas, maior é a proximidade entre a taxa real e a taxa desejada. Isto estava previsto na descrição do modelo (Seção 4.5.1): à medida que *N* cresce a taxa de descarte *r* tende a *N***r*. Percebe-se também que, embora o número de repetições seja elevado nos primeiros experimentos, a média de descarte ainda fica próxima da desejada.

Tabela 6.2: Cálculo do número de repetições necessárias para omissão no envio com taxa de descarte de 10% e margem de erro de 2,5%

Mensages enviadas	Medições realizadas (em número de mensagens descartadas)				Média		Desvio Padrão		Nº repetições IC (95%)
	1	2	3	4					
10^2 APPs	10	13	9	9	**10,25**	10,25%	1,89	**1,89%**	**628,92**
10^3 APPs	106	98	95	98	**99,25**	9,93%	4,72	**0,47%**	**41,65**
10^4 APPs	1018	1053	1020	972	**1015,75**	10,16%	33,29	**0,33%**	**19,81**
10^5 APPs	10005	10012	9984	9955	**9989,00**	9,99%	25,60	**0,03%**	**0,12**
10^6 APPs	100120	100396	100422	99965	**100225,75**	10,02%	221,11	**0,02%**	**0,09**

A título de exemplificação, foi realizado um teste seguindo a orientação para o número de repetições encontrado na Tabela 6.2 para um processo que envia 10^4 mensagens. Foram efetuadas 20 repetições do mesmo experimento e os resultados estão colocados na Tabela 6.3. A média de descarte ficou muito próxima do valor esperado e o desvio padrão obtido é bem baixo, confirmando uma pequena variação de cada experimento em relação a média.

Tabela 6.3: Descartes obitos por omissão no envio com uma taxa de 10%

Execução	Mensagens descartadas	Taxa de descarte
1	950	9,50%
2	937	9,37%
3	965	9,65%
4	976	9,76%
5	1001	10,01%
6	979	9,79%
7	1007	10,07%
8	1000	10,00%
9	996	9,96%
10	1020	10,20%
11	1005	10,05%
12	1008	10,08%
13	1004	10,04%
14	1031	10,31%
15	1027	10,27%
16	1055	10,55%
17	1031	10,31%
18	1034	10,34%
19	989	9,89%
20	1012	10,12%
Média	1001,35	10,01%
Desvio padrão	29,32804	0,29%
Intervalo de conf.	± 12,85 [988,5 – 1014,2]	

A partir destes testes conclui-se que a solução para omissão no envio funciona adequadamente de acordo com o modelo especificado, descartando mensagens em função da taxa de descarte configurada.

6.4.2 Omissão no recebimento

Todas as considerações anteriores referentes à classe de defeitos de omissão no envio também são válidas para a classe que permite a simulação de defeitos de omissão no recebimento. A única diferença é que, enquanto a classe `SendOmissionLayer` realiza os descartes no método `send()`, a classe `ReceiveOmissionLayer` os faz, do mesmo modo, no método `deliver()`. Isto significa que os dados obtidos na seção anterior também se aplicam a omissão no recebimento quando o método de Bernoulli é utilizado.

Para exemplificar o uso da camada de omissão no recebimento, optou-se por utilizar o método de descarte de Gilbert, descrito na seção 4.5.2. Foi utilizada a mesma estrutura da Figura 6.5. No entanto, neste exemplo, foram instanciados 10 processos: 1 `Server` e 9 `Clients`, visando verificar o comportamento da solução quando mais de um processo utiliza o descarte. A classe de omissão no envio foi configurada para não gerar descartes, restando apenas os descartes da classe de omissão no recebimento. A título de exemplificação foi realizada apenas uma execução para cada tipo de aplicação

Os parâmetros utilizados foram p=0,055 e q=0,728, o que resulta em uma taxa de descarte r=0,07 (7%)[3]. Os resultados da Tabela 6.4 demonstram que a taxa de descarte real ficou bastante próxima da prevista. No entanto, observa-se que quando o número de mensagens enviadas aumenta, há uma grande diminuição do desvio padrão (gráfico da Figura 6.6), o que representa uma menor variação do número de descarte de cada processo em relação a média obtida.

Tabela 6.4: Descarte das mensagens por omissão no recebimento ocasionadas pelo método de Gilbert com p=0,055 e q=0,728 (r=7%)

Processo	10^2 APPs		10^3 APPs		10^4 APPs		10^5 APPs		10^6 APPs	
p1	7	7,00%	83	8,30%	698	6,98%	7037	7,04%	70175	7,02%
p2	6	6,00%	75	7,50%	704	7,04%	7026	7,03%	70090	7,01%
p3	9	9,00%	51	5,10%	718	7,18%	7020	7,02%	70190	7,02%
p4	7	7,00%	65	6,50%	705	7,05%	7138	7,14%	70203	7,02%
p5	6	6,00%	60	6,00%	729	7,29%	7029	7,03%	70641	7,06%
p6	4	4,00%	77	7,70%	709	7,09%	7041	7,04%	70751	7,08%
p7	5	5,00%	64	6,40%	673	6,73%	6934	6,93%	70707	7,07%
p8	4	4,00%	68	6,80%	744	7,44%	7178	7,18%	70056	7,01%
p9	8	8,00%	58	5,80%	694	6,94%	7094	7,09%	70152	7,02%
Média	**6,22**	**6,22%**	**66,78**	**6,68%**	**708,22**	**7,08%**	**7055,22**	**7,06%**	**70329,44**	**7,03%**
DesvPad	**1,72**	**1,72%**	**10,12**	**1,01%**	**20,57**	**0,21%**	**71,97**	**0,07%**	**282,89**	**0,03%**

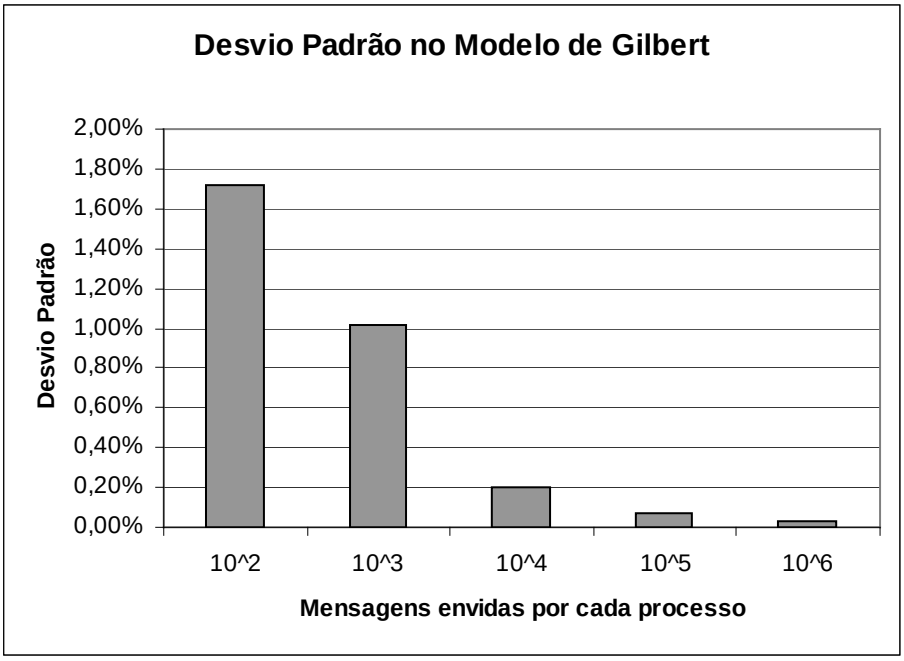


Figura 6.6: Variação do desvio padrão nos descartes por omissão no recebimento

A probabilidade de ocorrência de rajadas (Tabela 6.5) foi calculada utilizando a equação (2) da seção 4.5.2. As rajadas da simulação com 10^5 mensagens enviadas por processo foram registradas durante a simulação e estão expostas na Tabela 6.6. É possível observar ocorre uma maior quantidade de descartes em rajadas de tamanho 1. Esta quantidade diminui a medida que o tamanho da rajada aumenta, confirmando o que está apresentado na Tabela 6.6.

[3] Estes valores foram retirados do trabalho de Ni, Turletti e Fu (2002). Neste mesmo trabalho existem outros valores para p e q que resultam em taxas de descarte diferenciadas.

Tabela 6.5: Probabilidade de ocorrência de rajadas para q=0,728

k	Probabilidade	
1	0,728	72,9%
2	0,198	19,8%
3	0,054	5,4%
4	0,015	1,5%
5	0,004	0,4%
6	0,001	0,1%

Tabela 6.6: Rajadas geradas pelo método de Gilbert com p=0,055 e q=0,728

Processos	Tamanho das Rajadas					
	0	1	2	3	4	5
p1	92963	6097	428	24	3	0
p2	92974	6048	430	38	1	0
p3	92980	6144	394	28	1	0
p4	92862	6208	404	38	2	0
p5	92971	6139	393	29	3	1
p6	92959	6136	410	27	1	0
p7	93066	6001	418	31	1	0
p8	92822	6205	436	31	2	0
p9	92906	6151	424	29	2	0
Total	**836503**	**55129**	**3737**	**275**	**16**	**1**
Média	**92944,78**	**6125,44**	**415,22**	**30,56**	**1,78**	**0,11**

6.4.3 Omissão geral

Visando avaliar o comportamento das classes de omissão quando utilizadas em conjunto, foram realizadas experiências em que as duas classes realizavam descartes simultaneamente. A taxa de 10% foi mantida para cada uma das classes, o que ocasionou uma taxa de descarte real de aproximadamente 20%, como demonstrado na Tabela 6.7. O método de descarte utilizado foi o de Bernoulli. Como pode ser observado, a quantidade de mensagens descartadas por omissão no recebimento tende a ser menor que o número de mensagens descartadas por omissão no envio. Isto aconteceu porque o descarte no recebimento é realizado sobre as mensagens de reconhecimento (ACK). Uma vez que algumas mensagens APP foram descartadas ainda no envio e, portanto, não chegaram ao Server, estas não geraram mensagens de reconhecimento. Como exemplo, o processo 1 enviou 1000 mensagens do tipo APP. Como 98 destas foram descartadas no envio, apenas 902 mensagens ACK foram submetidas ao descarte no recebimento. Assim, embora o número de mensagens descartadas seja menor, a porcentagem de descartes foi equivalente tanto no envio quanto no recebimento, já que um número menor de mensagens de retorno foi gerado.

O gráfico da Figura 6.7 mostra que as classes de omissão descartam mensagens proporcionalmente quando utilizadas em conjunto. As oscilações entre descartes no envio e no recebimento são ocasionadas pela característica aleatória do método de descarte utilizado.

Tabela 6.7: Descartes ocasionados por omissão geral com taxa de 10%

Processos	APPs Enviadas	Descartes no envio		Descartes no recebimento		Total de descartes	
p1	1000	98	9,80%	96	10,64%	194	20,44%
p2	1000	111	11,10%	93	10,46%	204	21,56%
p3	1000	96	9,60%	92	10,18%	188	19,78%
p4	1000	92	9,20%	85	9,36%	177	18,56%
p5	1000	104	10,40%	96	10,71%	200	21,11%
p6	1000	107	10,70%	83	9,29%	190	19,99%
p7	1000	110	11,00%	89	10,00%	199	21,00%
p8	1000	94	9,40%	87	9,60%	181	19,00%
p9	1000	119	11,90%	81	9,19%	200	21,09%
Média		**103,44**	**10,34%**	**89,11**	**9,94%**	**192,56**	**20,28%**
Desv. Pad.		**9,08**	**0,91%**	**5,51**	**0,60%**	**9,28**	**1,03%**

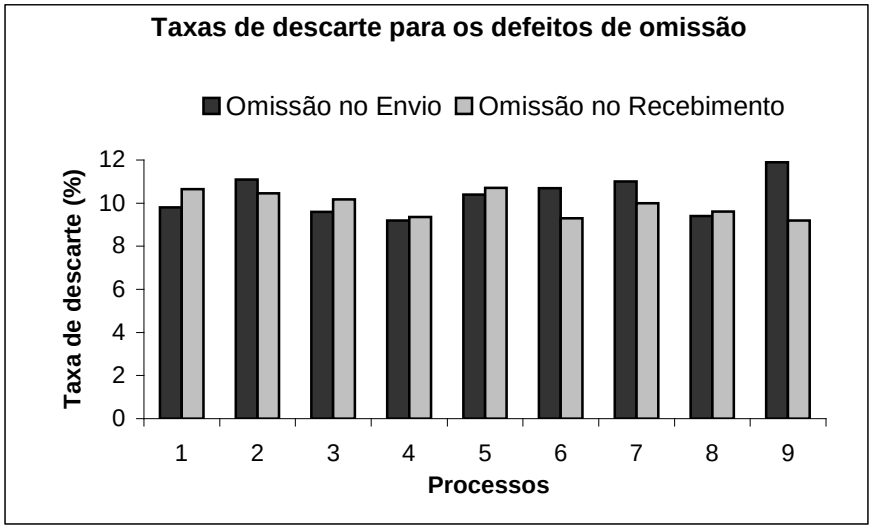


Figura 6.7: Gráfico comparativo com as taxas de descarte no envio e no recebimento

6.5 Estudo de Caso para Defeitos de Colapso

Como estudo de caso para os defeitos de colapso foi utilizada a aplicação descrita na seção 6.1 em conjunto com o algoritmo de *checkpointing* da seção 6.2 e com o detector de defeitos da seção 6.3. Embora estes algoritmos tenham sido utilizados em parte nos estudos de caso de omissão e rede, eles foram desenvolvidos especificamente para testar e exemplificar as classes de colapso. Entretanto, como o objetivo principal deste estudo é avaliar o funcionamento e o impacto da utilização das classes de colapso nas aplicações foram realizados, num primeiro passo, experimentos utilizando apenas a aplicação em conjunto com as classes de colapso. Os demais algoritmos serão utilizados em um estudo de caso posterior.

Os primeiros testes realizados utilizaram a emulação. O cenário continha três processos, um em cada máquina. Foram utilizadas as seguintes máquinas do laboratório de TF da UFRGS: *jaguar.inf.ufrgs.br* (`Server`), *bentley.inf.ufrgs.br* e *maverick.inf.ufrgs.br* (`Clients`). A máquina *chevy.inf.ufrgs.br* foi utilizada para gravar os *logs*. Cada máquina é composta por Processador AMD XP 2400+, memória de 512 MB, disco de 80 GB e possuem o sistema operacional Ubuntu 5.04 instalado.

Como exemplo, a Figura 6.8 mostra o instante do colapso de um dos *logs* gerados pelo processo com defeito (instanciado na máquina *bentley*). O colapso foi configurado para ocorrer no tempo 1000*ms*. No entanto, o mesmo só aconteceu no tempo 1255*ms*. O retorno também não aconteceu no tempo configurado, que era 1100*ms*. Isto ocorre porque na emulação o tempo é real. Assim, mesmo que o colapso seja invocado no tempo 1000*ms*, tanto a execução da rotina que o inicia quanto o tempo de salvamento no *log* consomem tempo de processamento. Embora isto possa ser inconveniente em alguns casos, não deixa de ser uma característica interessante, uma vez que oferece execuções menos determinísticas.

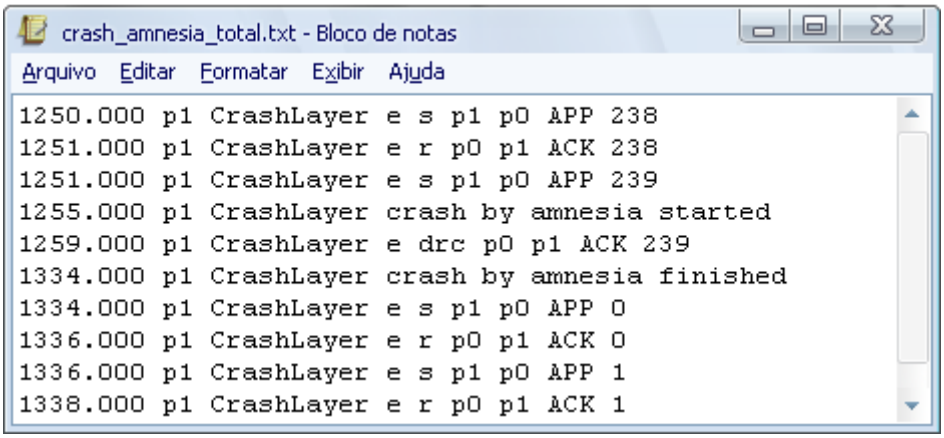


Figura 6.8: Arquivo de log gerado na emulação de colapso com amnésia total

Na figura também é possível notar o descarte da mensagem 239, ocasionado pelo colapso. Os demais testes realizados por emulação estão disponíveis no CD-ROM do Apêndice B.

Por curiosidade foram executadas algumas emulações visando observar o impacto da escrita dos *logs* no tempo de execução (Figura 6.9). Verificou-se que as aplicações NET e HIGH apresentam maiores diferenças. Isto se dá devido ao grande número de mensagens enviadas e que também são escritas no *log*. Além disso, as aplicações BAL, CPU e LOW possuem tempos de espera fixos após o envio da mensagem, o que determina um tempo mínimo para cada execução.

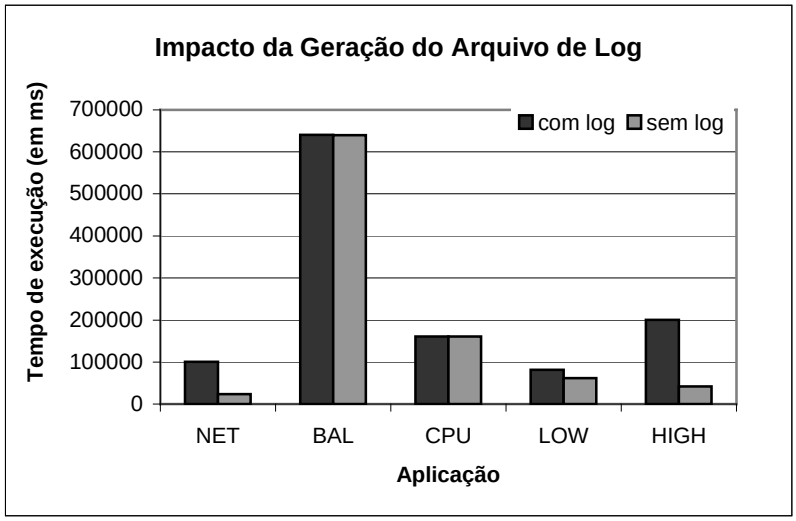


Figura 6.9: Impacto da geração dos arquivos de log na emulação

Estas emulações também foram úteis para determinar o tempo médio de envio das mensagens na rede real, subtraindo-se o tempo de chegada da mensagem de ACK do tempo de envio da mensagem de aplicação (APP). Este tempo médio foi utilizado nas simulações e corresponde a 0,58*ms*.

As simulações seguiram a mesma configuração da emulação. Primeiramente, foram realizadas testes com a aplicação sendo executada normalmente, isto é sem qualquer tipo de inserção de defeito. Em seguida, foram executados os testes com os três tipos de colapso: pausa, amnésia parcial e amnésia total. O colapso por parada não foi incluído nestes testes porque não possui retorno e, portanto, não permite que a aplicação desenvolvida termine seu processamento. Neste caso não seria possível realizar comparações com os demais tipos de colapso.

A simulação contou com três processos: um `Server` (*p0*) e dois `Clients` (*p1* e *p2*) O `Client 1` (*p1*) foi escolhido para receber os defeitos de colapso. Em todos os casos, o colapso é iniciado no tempo 1000*ms* e finalizado no tempo 1100*ms*. Na simulação proposta não é difícil prever o resultado do colapso no tempo de execução da aplicação. Para colapsos por pausa, basta incluir o tempo de pausa no tempo final, uma vez que o processo não precisa retransmitir mensagens, a não ser que haja a necessidade de garantia de entrega das mensagens enviadas, o que não é feito pela aplicação em questão. No entanto, todas as mensagens recebidas pelo processo durante o período de defeito são descartadas. Isto vale para todas as classes de colapso desenvolvidas.

No experimento com colapso com amnésia parcial o salvamento das mensagens enviadas foi programado para ocorrer a cada 150*ms*. Isto significa que se o defeito acontecer antes dos primeiros 150*ms*, todas as mensagens serão reenviadas. Se acontecer entre os instantes 150*ms* e 300*ms*, apenas as mensagens enviadas após o último salvamento serão reenviadas e assim sucessivamente. Uma vez que o colapso ocorreu no instante 1000*ms*, apenas as mensagens enviadas após o instante 900*ms* foram reenviadas. O tempo de reenvio destas mensagens será a diferença no tempo final.

No último experimento, o colapso com amnésia total faz com que o processo volte a reenviar todas as mensagens enviadas antes do defeito. Este tempo de reenvio também será acrescentado ao tempo final. Nos dois casos de amnésia, o tempo em que o processo esteve parado é automaticamente acrescentado no tempo total.

Os valores obtidos com estes cenários estão dispostos na Tabela 6.8 e ilustrados no gráfico da Figura 6.10. Como previsto, a amnésia total consome o maior tempo, já que um grande número de mensagens precisa ser retransmitido.

Tabela 6.8: Impacto da ocorrência de colapso na aplicação simulada

Aplicação	Execução Normal	Execução com Colapso		
		Pausa	Amnésia Parcial	Amnésia Total
NET	69900,58	69999,85	70301,74	71100,34
BAL	126602,58	126701,70	126800,70	127811,75
CPU	71658,42	71752,73	71858,74	81899,70
LOW	35040,58	35139,61	35342,64	36240,24
HIGH	3000,58	3100,58	3202,78	4000,53

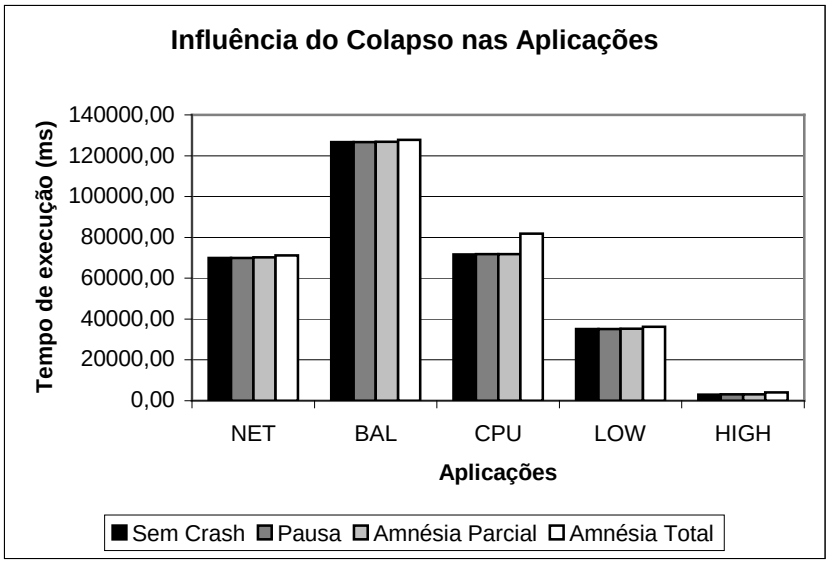


Figura 6.10: Gráfico do impacto do colapso na aplicação

O número de mensagens retransmitidas por cada tipo de colapso e em cada cenário está colocado na Tabela 6.9. A aplicação HIGH retransmite um grande número de mensagens, por possuir a característica de enviar mais mensagens e de forma mais rápida. Assim, embora o colapso aconteça no mesmo tempo para todas as aplicações, aquelas que enviam mais mensagens por unidade de tempo são mais prejudicadas. Através destes resultados verifica-se também o comportamento de cada aplicação, já descrito na seção 6.1. As aplicações NET e LOW utilizam mais rede que a aplicação CPU. A aplicação BAL se comporta de forma intermediária e a aplicação HIGH faz uso extremo da rede.

Tabela 6.9: Quantidade de menagens retransmitidas nos colapsos com amnéisa

Aplicação	Amnésia Parcial	Amnésia Total
NET	87	858
BAL	31	350
CPU	14	154
LOW	87	941
HIGH	20000	199999

Depois de analisados os efeitos do colapso na execução da aplicação sintética, foi realizado um experimento simulado contendo, além da aplicação, o algoritmo de *checkpointing* e o detector de defeitos. A pilha de camadas utilizada segue a estrutura da Figura 6.11. Foram isoladas as camadas de aplicação (`Server` e `Client`) e a camada de *checkpointing*. Isto significa que, uma vez iniciado o colapso, os módulos detectores continuam comunicando-se. Desta forma, quando um módulo detector suspeitar de um colapso, o mesmo conseguirá avisar todos os demais módulos detectores sobre a suspeita, até mesmo se o processo suspeito estiver na mesma máquina.

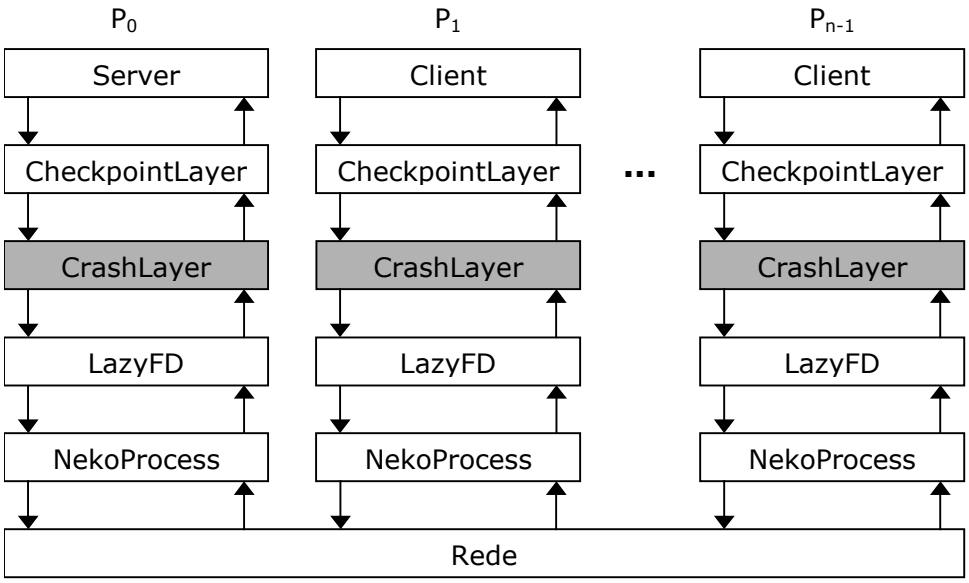


Figura 6.11: Pilha de camadas utilizada no estudo de caso para defeitos de colapso

No cenário proposto, o detector instalado na mesma pilha do processo em colapso detectará o defeito antes dos demais, uma vez que o detector verifica primeiramente o processo local, o que é feito de maneira mais eficiente, pois não necessita de transmissão pela rede. O tempo de detecção do defeito depende do intervalo de tempo em que o detector verifica o estado da aplicação e do *timeout* das mensagens em trânsito. Quanto menor o intervalo, mais rápida é a detecção, embora ocorra o aumento do consumo de recursos da rede para envio de mensagens de confirmação (`AREQ_IMLIVE`).

Na simulação executada, cada processo envia 250 mensagens do tipo APP ao servidor e aguarda pelas mensagens de reconhecimento (`ACKs`). Para cada mensagem APP recebida pelo servidor, uma mensagem de reconhecimento (`LAZY_ACK`) é enviada pelo detector ao cliente emissor. Além disso, como o servidor envia ACKs no nível da aplicação, outra mensagem `LAZY_ACK` é enviada pelo cliente ao servidor. As mensagens `LAZY_ACK` são utilizadas pelo detector para identificar possíveis defeitos.

A camada de *checkpointing* intercepta todas as mensagens enviadas pelo cliente e as inclui na lista de mensagens em trânsito. Estas mensagens são retiradas da lista à medida que os `ACKs` são recebidos do servidor.

Para testar todos os mecanismos envolvidos na detecção e recuperação foram configurados diferentes tipos de colapso. A diferença em cada um deles é basicamente o estado em que a aplicação retorna após o defeito. O processo cliente *p1* foi escolhido para sofrer colapsos por pausa, parada, amnésia parcial e amnésia total. Em todos os casos, o defeito inicia no tempo 199*ms* e finaliza no tempo 230*ms*. Uma vez que o detector foi configurado para fazer verificações a cada 200*ms*, o mesmo suspeitou do processo *p1* e avisou aos demais detectores sobre a suspeita. Cada detector se responsabiliza por avisar o mecanismo de *checkpointing* e este, por sua vez, se encarrega de avisar a aplicação, solicitando que a mesma que pare sua execução.

A próxima etapa é identificar o ponto de recuperação comum a todas as aplicações. Esta fase é iniciada tão logo o detector verifique o final do defeito. Esta detecção é feita através de sondagem do processo defeituoso. Isto significa que para detectar o fim do colapso o processo defeituoso precisa voltar a funcionar. É neste ponto que os tipos de defeito inseridos diferem uns dos outros. Para defeitos de pausa, quando a aplicação volta a funcionar a próxima mensagem enviada é a imediatamente posterior a última mensagem enviada antes do defeito. Para colapso por amnésia total, a mensagem

enviada no retorno é a primeira mensagem, com identificador zero. No colapso por amnésia parcial, a mensagem enviada é a imediatamente posterior ao último salvamento de estado da aplicação

No entanto, para este exemplo de aplicação, o estado em que a aplicação retorna após o defeito não é relevante, uma vez que o mecanismo de *checkpointing* se encarregará de sincronizar os processos em um estado consistente. Nos testes realizados, o mecanismo de *checkpointing* foi configurado para executar salvamentos de estado a cada 100.0*ms*, o último ponto de restauração consistente é o ponto gerado no instante 100.0*ms*. As aplicações foram reiniciadas e todas as mensagens enviadas após o ponto de restauração foram reenviadas. Como o defeito ocorreu no tempo 199*ms*, isto é, pouco antes do segundo ponto de restauração que ocorreria no tempo 200*ms*, um grande número de mensagens foi reenviado. Este número diminuiu quando o intervalo de salvamento foi alterado para 50.0*ms*.

No estudo de caso utilizado, o detector não foi isolado pelo colapso. No entanto, nada impede que a camada de colapso seja colocada abaixo do detector. Nesse caso, se ocorrer um colapso e o detector local suspeitar do processo ele tentará avisar os demais detectores enviando uma mensagem de detecção de colapso, que será descartada pela camada de colapso. Entretanto, haverá um momento em que outro módulo detector de outra máquina suspeitará do processo, visto que as mensagens de aplicação não serão reconhecidas, e avisará aos demais.

6.6 Estudos de Caso para Defeitos de Rede e Particionamento

Como já descrito nas seções anteriores, os tipos de defeito propostos para simulação de defeitos no nível de rede são a quebra de enlace, o descarte de mensagens e o particionamento de grupos de processos. Assim, foram realizados alguns testes visando exemplificar o funcionamento de um cenário para cada tipo de defeito proposto.

6.6.1 Quebra de enlace

A quebra de enlace é especificada entre pares de processos. Para exemplificar o uso deste tipo de defeito, foi utilizada a mesma estrutura definida para o exemplo de colapso, retirando-se a camada de *checkpoiting*. Com isso, foi possível verificar o comportamento do algoritmo de detecção de defeitos frente a esta classe de defeitos. A diferença entre o defeito de enlace e o defeito de colapso é que, no defeito de rede do tipo quebra de enlace, toda a pilha do processo isolado é desconectada da rede. Isso faz com que cada processo isolado suspeite do outro.

Como exemplo criou-se um ambiente simulado contendo 3 processos: um coordenador (*p0*) e 2 clientes (*p1* e *p2*). O protocolo de rede utilizado foi o UDP, que não fornece retransmissão de mensagens perdidas. Foi realizada uma quebra de enlace entre os processos *p0* e *p1*, iniciando no instante 26,5*ms* e finalizada no tempo 33*ms*. É possível verificar na Figura 6.12 que o processo *p0* enviou uma mensagem de ACK para o processo *p1* no tempo 26,1*ms*. Ao receber a mensagem de *p0* o detector em *p1* tenta enviar um ACK ao detector em *p0*. Entretanto, devido à quebra do enlace, esta mensagem é descartada e ocasiona a suspeita de *p1* em *p0*. Além disso, *p1* tenta enviar uma mensagem de aplicação (APP) para *p0* no tempo 27,1*ms*, que também é descartada pela rede. Isso fará com que o detector em *p1* suspeite de *p0*, já que não receberá ACK para esta mensagem. A suspeita, tanto de *p1* quanto de *p0*, se deu no tempo 30*ms*, uma vez que os detectores foram configurados para realizar verificações a cada 30*ms*. Instaurada a suspeita, *p0* e *p1* enviaram mensagens de notificação (NOTIFY_CRASH) da suspeita para todos os processos, incluindo o processo local. Cada processo que

recebeu a mensagem de notificação respondeu com uma mensagem de confirmação (ANOTIFY_CRASH). É possível verificar que nem *p0* e nem *p1* receberam as mensagens de notificação um do outro, pois o enlace ainda está com desfeito.

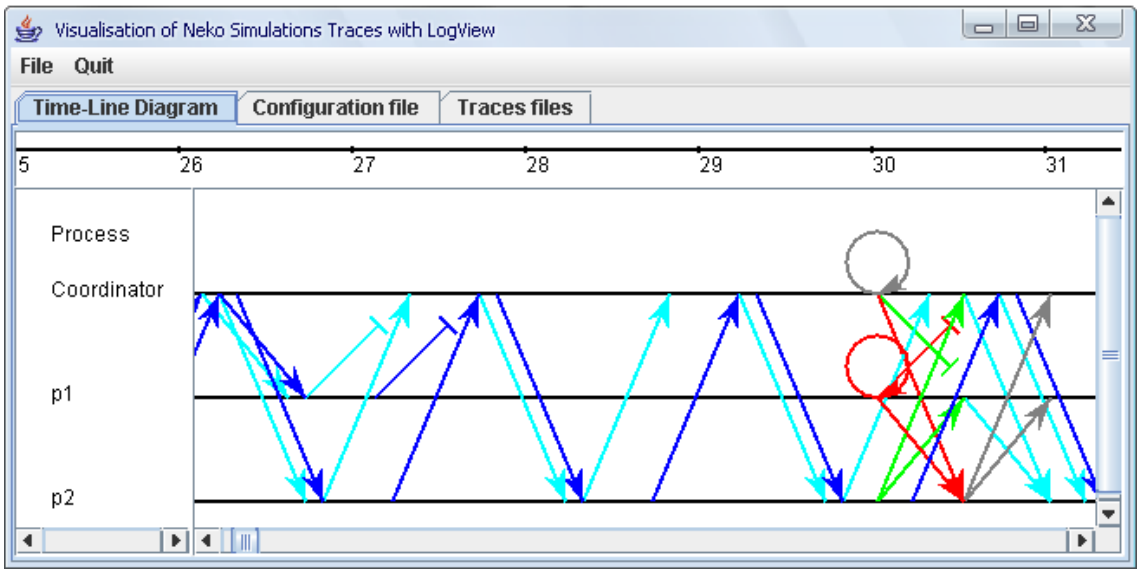


Figura 6.12: Comportamento do algoritmo de detecção frente a uma quebra de link

A segunda etapa do detector visa identificar a recuperação do(s) processo(s) suspeito(s). O processo *p0* inicia a etapa de sondagem de *p1* através do envio de mensagens de PING. O mesmo acontece com o processo *p1* em relação à *p0*. O detector permanece nesta fase até que uma resposta (ACK) seja recebida do processo suspeito. Na Figura 6.13 a primeira resposta foi recebida no tempo 35.0. Assim, *p0* e *p1* enviam no tempo 36.0 mensagens de RESTART a todos os processos, inclusive aos processos locais, sinalizando que a suspeita terminou.

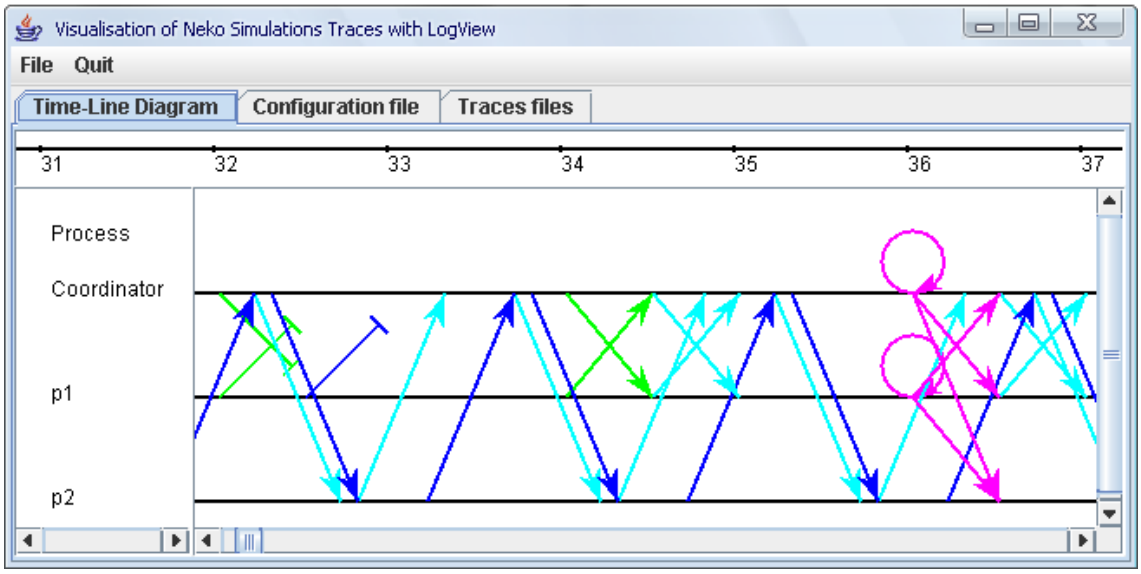


Figura 6.13: Fase de identificação do fim do defeito de link

Através deste exemplo foi possível verificar o correto funcionamento da inserção de defeito de enlace entre pares de processos e as implicações que o mesmo pode ocasionar nas aplicações sendo simuladas.

6.6.2 Descarte de mensagens

O descarte de mensagens no nível de rede utiliza as mesmas políticas disponibilizadas para omissão. A diferença é que na rede as mensagens são descartadas sem levar em consideração o processo que as envia. Assim, para políticas que fazem uso de probabilidade, a possibilidade de descarte é maior para processos que enviam mais mensagens.

6.6.3 Particionamento

Para simular o particionamento dos nós da rede é necessário definir grupos. A Figura 6.14 (gerada com auxilio da ferramenta LogView) exemplifica o uso da solução desenvolvida. Foram definidos dois grupos. O primeiro grupo inclui os processos *p0* (*Coordinator*), *p1* e *p2*. O segundo grupo contém os processos *p3*, *p4* e *p5*. O algoritmo utilizado é aquele disponível no pacote de exemplos do Neko e simula o padrão de comunicação no consenso de Lamport (1978). O processo coordenador envia uma mensagem a todos os processos. Cada processo que recebe a mensagem do coordenador envia uma outra mensagem a todos os demais processos, incluindo o coordenador. Entretanto, no instante 5.0 foi iniciado o particionamento dos grupos e as mensagens enviadas por processos do grupo 1 não puderam ser entregues aos processos do grupo 2 e vice-versa.

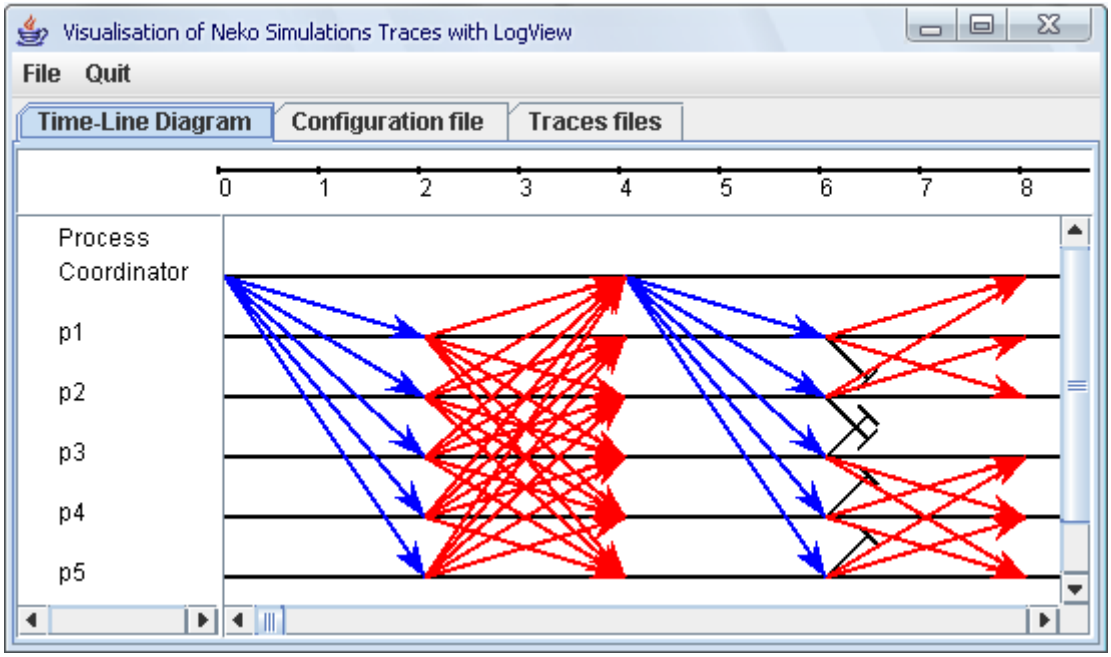


Figura 6.14: Exemplo de ocorrência de particionamento nos nós da rede

Este teste, embora extremamente simples, ilustra e atesta o funcionamento da classe de defeitos de particionamento, que nada mais é do que uma extensão da classe de defeitos de quebra de enlace.

7 CONSIDERAÇÕES FINAIS

7.1 Conclusões

O desenvolvimento de novas ferramentas para o estudo de sistemas distribuídos favorece a pesquisa e desenvolvimento de novos mecanismos para superação dos problemas existentes. Além disso, os simuladores são ferramentas bastante utilizadas para obter bons resultados a um baixo custo e em tempo reduzido.

Neste trabalho, procurou-se ampliar o modelo de defeitos do *framework* Neko, permitindo assim, uma maior flexibilidade para a pesquisa na área de tolerância a falhas. O levantamento do suporte disponível no Neko e o desenvolvimento das novas funcionalidades tende a favorecer futuros usuários do *framework*, proporcionando um mecanismo de pesquisa mais amplo e diversificado.

De acordo com as classes de defeito estudadas neste trabalho, foram inseridos mecanismos para a simulação de defeitos de omissão de mensagens no envio e no recebimento; colapsos com pausa, parada, amnésia total e amnésia parcial; e defeitos de rede, que incluem quebra de enlace, descarte de pacotes e particionamento de grupos. Com este novo suporte de simulação é possível realizar diversos tipos de teste com inserção de defeitos em algoritmos distribuídos, favorecendo a verificação e validação dos mesmos.

Uma das deficiências do Neko é a pouca documentação disponível. Como uma tentativa de facilitar o uso da ferramenta pela comunidade pesquisadora criou-se um pequeno manual que descreve a instalação, modos de execução dos experimentos, estrutura do *framework* e seus principais componentes. O manual foi inserido como apêndice no trabalho.

Como o Neko é um projeto em desenvolvimento, novas atualizações e versões estão constantemente sendo disponibilizadas. Assim, a última versão do Neko (v1.0 alpha), sofreu grandes modificações estruturais, principalmente no que diz respeito a estrutura de camadas do *framework*. Nesta última versão, cada camada funciona como um micro-procolo e consegue se comunicar diretamente, isto é, sem que as mensagens tenham que ser repassadas por camadas inferiores. Para atender a estas modificações, as soluções apresentadas neste trabalho precisam ser adaptadas. Porém, por questões de tempo e escopo, tais modificações foram sugeridas como trabalhos futuros.

7.2 Uma Contribuição a Outros Desenvolvedores

O Grupo de Tolerância a Falhas da UFRGS tem trabalhado nos últimos anos com injeção de falhas por software, sendo o trabalho de Jaques-Silva (2005) um dos mais recentes. No âmbito de contribuição à área de simulação de defeitos, a pesquisa

realizada por Trindade (2003) procurou investigar o uso do simulador de redes NS (*Network Simulator*) como ambiente para simulação de sistemas distribuídos, particularmente em cenários sujeitos à ocorrência de defeitos. No entanto, o trabalho identificou dificuldades para modelagem e utilização do NS em sistemas distribuídos. Por esta razão, realizou-se um trabalho de levantamento de simuladores disponíveis na área, voltado a sistemas distribuídos e que tivessem código aberto e possibilidade de adaptação. Este levantamento foi apresentado em forma de artigo na disciplina de Tolerância a Falhas e faz referência a duas ferramentas de simulação de algoritmos distribuídos. Uma destas ferramentas é o *framework* Neko. A partir de então, foi estabelecido contato com os desenvolvedores do Neko e levantou-se a questão de incluir novas funcionalidades para permitir a simulação de defeitos em algoritmos distribuídos utilizando a estrutura já definida do *framework*.

O primeiro passo para realizar estas inclusões foi levantar quais tipos de defeitos seriam trabalhados. Para tanto, foram consultados os autores da seção 2. A partir da classificação destes autores, foi estabelecido que seriam criados mecanismos para simular defeitos de omissão de mensagens, colapso de processos, rede e particionamento.

Uma vez que no Neko as mensagens são transmitidas através das camadas da aplicação, a solução para criar omissão de mensagens no envio e no recebimento foi instalar uma camada que intercepta as mensagens e realiza o descarte apropriadamente. Neste ponto, surgiu a necessidade de se definir políticas de descarte, isto é, quando descartar e quando repassar uma mensagem. Recorrendo novamente a literatura, foram estudados e implementados quatro modelos de descarte: dois utilizando probabilidade e com comportamento determinístico (seção 4.5).

A segunda classe de defeitos abordada foi o colapso de processos. Optou-se pela definição de Cristian (seção 2.1) por ser mais completa. Nesta etapa, o maior problema era conseguir parar a aplicação durante o colapso, isto significa em termos de implementação, bloquear o objeto Java que executa o código da aplicação sendo simulada no processo com defeito. A primeira tentativa para conseguir resolver este problema envolveu o uso de *Java Platform Debugger Architecture* (JPDA) que é um pacote que pode ser utilizado como suporte para o desenvolvimento de ferramentas de depuração para a plataforma Java. Uma vez conectado a aplicação o depurador tem acesso a diversas informações sobre a VM em execução. É possível listar as classes sendo utilizadas, *threads* em execução, fila de eventos, etc. Além disso, o depurador tem a disposição métodos para suspender, reiniciar e terminar a execução da VM do depurado. O mesmo pode ser feito para as *threads* em execução. É possível ainda inserir *breakpoints* e *watchpoints* em qualquer ponto de execução da aplicação e obter valores dos atributos definidos. No entanto, esta solução foi abandonada porque fugia do objetivo principal do trabalho, que é estender o *framework* e disponibilizar novas funcionalidades que possam ser utilizadas e expandidas dentro da própria ferramenta e não utilizando uma ferramenta auxiliar. Além disso, descobriu-se uma forma mais eficaz e simplificada de realizar o bloqueio da aplicação utilizando o próprio escalonador do Neko. Restava ainda, para os tipos de colapso por amnésia e amnésia parcial, encontrar uma forma de salvar o estado da aplicação antes do colapso e poder restabelecer o estado após o defeito. A solução encontrada foi utilizar o pacote `java.lang.reflect`, que permite a leitura e escrita de atributos e a execução de métodos em objetos Java em tempo de execução.

A última classe de defeitos abordada foi a de rede, incluindo o particionamento. Nesta etapa surgiram dois problemas: o primeiro é que a estrutura original do Neko não permite incluir camadas entre a rede e o `NekoProcess`. Se isto fosse possível, poderia

ser criada uma camada que fizesse a inserção de defeitos de rede logo após o envio da mensagem pelo processo. No entanto, esta solução exigiria grandes alterações estruturais na ferramenta, além de modificar a forma como a pilha de processos é criada no início da simulação. Assim, uma solução mais simples foi criar um tipo de rede com suporte a defeitos de rede e particionamento que encapsula uma outra rede qualquer, já incluída no Neko. Desta forma, o defeito é inserido de forma transparente e as mensagens que devem ser enviadas sem defeito, são transmitidas normalmente utilizando a rede escolhida.

Alguns resultados parciais foram submetidos ao WTD 2005 (*Third Workshop on Theses and Dissertations in Dependable Computing*) (RODRIGUES e JANSCH-PÔRTO, 2005), e os resultados finais foram submetidos ao WTF 2006 (VII *Workshop* de Testes e Tolerância a Falhas).

7.3 Trabalhos Futuros

Como trabalhos futuros são propostos:

- estudar e implementar novas políticas de descarte de mensagens para serem utilizadas nas simulações de defeitos de omissão e descarte de mensagens pela rede de comunicação.
  Na seção 4.5 estão descritas algumas políticas de descarte que não foram abordadas no trabalho. Há outras descritas na bibliografia citada que não foram referenciadas no texto. Além disso, a distribuição aleatória utilizada foi a uniforme. Sugere-se o estudo e inclusão de outras distribuições, como a normal, hiperbólica, de Pareto, entre outras.
- Estudar o funcionamento do SSFNet e verificar quais as possíveis modificações que podem ser realizadas para permitir a adaptação das soluções para simulação de defeitos propostas neste trabalho em cenários que utilizam o suporte de topologia oferecido pelo SSFNet.
  A possibilidade de incluir topologia nas simulações de defeitos favorece a qualidade dos resultados, uma vez que aproxima a simulação da situação real. Para tanto, é preciso estudar as classes Java utilizadas na integração do SSFNet ao Neko bem como as próprias classes que integram o simulador para realizar, se possível, as adaptações necessárias para utilizá-lo na simulação dos defeitos descritos neste trabalho.
- Incluir mecanismos para simulação de defeitos de resposta e temporização, descritos por Cristian (1991);
  A ocorrência de atrasos, principalmente nas redes de comunicação, é freqüente. Estudar o comportamento de uma aplicação frente a este tipo de defeito é muito importante no contexto de tolerância a falhas.
- Conduzir novos experimentos com inserção de defeitos, visando verificar de forma mais aprofundada a validade e utilidade das soluções propostas.
  A proposta é executar as aplicações em conjunto com as classes de defeito desenvolvidas modificando os parâmetros de ocorrência dos defeitos, número de processos envolvidos, tipo de rede utilizada, tanto em simulações quanto nas emulações.
- Adaptar as soluções para simulação de defeitos apresentadas neste trabalho para serem utilizadas pela nova versão (1.0 alpha) do Neko;
  A versão alpha do Neko apresenta uma nova estrutura de comunicação, que dispensa a necessidade de transmitir as mensagens através da pilha de camadas em cada processo. Como as soluções propostas pelo presente trabalho utilizam

esta característica de pilha para interceptar as mensagens e gerar os defeitos, é preciso identificar quais as modificações necessárias para adaptar as soluções, evitando assim que as novas versões não possam utilizar os mecanismos desenvolvidos.

- Atualizar o manual desenvolvido para o Neko (Apêndice A Manual do Neko) para refletir as alterações na nova versão do *framework* (v.alpha).
  Todo o texto do manual está voltado para a versão 9.0 do Neko. Entretanto, nem todos os mecanismos sofreram alterações na versão alpha. Assim, a proposta é alterar o texto para incluir/alterar a nova estrutura e funcionalidades.
- Verificar se são necessárias adaptações na ferramenta LogView para criar a visualização dos *logs* gerados pela nova versão (v.alpha), que inclui agora o identificador do protocolo gerador do evento.

REFERÊNCIAS

BALBINOT, J. I. **Implementação de um serviço configurável de detecção de defeitos**. 2006. 80 f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre, 2006. (Em preparação)

BIRMAN, K. P. **Building Secure and Reliable Network Applications**. Greenwich: Manning, 1996.

BULIGON, C.; CECHIN, S. L.; JANSCH-PÔRTO, I. Implementing Rollback-Recovery Coordinated Checkpoints. In: INTERNATIONAL SCHOOL AND SYMPOSIUM ADVANCED OF DISTRIBUTED SYSTEMS, ISSADS, 5., 2005. **Proceedings…** Berlin: Springer-Verlag, 2005. p. 246-257.

CECHIN, S. L. **Protocolo de Recuperação por Retorno, Coordenado, não Determinístico**. 2002. 844 f. Tese (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

CRAIG, D. **Extensible Hierarchical Object-Oriented Logic Simulation with an Adaptable Graphical User Interface**. 1996. 197 p. Master Thesis (Master of Science) – Department of Computer Science, University of Newfoundland, Newfoundland.

CRISTIAN, F. Understanding Fault-tolerant Distributed Systems. **Communications of the ACM**, New York, v. 34, n.2, p. 57-78, Feb. 1991.

FALAI, L; BONDAVALLI, A.; DI GIANDOMENICO, F.. **Quantitative Evaluation Using Neko Tool:** NekoStat Extensions. Firenze: [s.n.], 2004. DSI Technical Report.

FETZER, C.; RAYNAL, M.; TRONEL, F.. An Adaptive Failure Detection Protocol. In: IEEE PACIFIC RIM INTERNATIONAL SYMPOSIUM ON DEPENDABLE COMPUTING, 8., 2001. **Proceedings…** [S.l.:s.n.], 2001. p.146-153.

JACQUES-SILVA, G. **Injeção Distribuída de Falhas para Validação de Dependabilidade de Sistemas Distribuídos de Larga Escala**. 2005. 79 f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

JAIN, R. **The art of computer systems performance analysis:** techniques for experimental design, measurement, simulation, and modeling. New York: John Wiley, 1991.

JALOTE, P. **Fault Tolerance in Distributed Systems**. Englewood Cliffs: Prentice Hall, 1994.

JAVADOC. **Why are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit Deprecated?** Disponível em: <http://java.sun.com/j2se/1.5.0/docs/api/>. Acesso em: ago. 2005.

JIANG, W.; SCHULZRINNE, H.. Modeling of Packet Loss and Delay and their Effect on Real-time Multimedia Service Quality. In: INTERNATIONAL WORKSHOP ON NETWORK AND OPERATING SYSTEM SUPPORT FOR DIGITAL AUDIO AND VIDEO, NOSSDAV, 10., 2000. **Proceedings…** [S.l.:s.n.], 2000.

LAMPORT, L. Time, Clocks, and the Ordering of Events in a Distributed System. **Communications of the ACM**, New York, v. 21, n.7, p.558-565, 1978.

LAPRIE, J.. **Dependability of Computer Systems:** From Concepts to Limits. In: IFIP INTERNATIONAL WORKSHOP ON DEPENDABLE COMPUTING AND ITS APPLICATIONS, 1998. **Proceedings…** Johannesburg: University of Witwathersrand, 1998. p. 108-126.

MATSUSHITA, T. **Integrating the Neko Distributed Programming Framework with SSFNet Network Simulator**. 2005. Master thesis – School of Information Science, JAIST, Japan.

MARKOVSKI, V. **Simulation and Analysis of Loss in IP Networks**. 2000. 87 f. Master Thesis (Master of Applied Science) – School of Engineering Science, Simon Fraser University, Canadá.

NEIGER, G.; TOUEG, S. Automatically Increasing the Fault-tolerance of Distributed Systems. In: ANNUAL ACM SYMPOSIUM ON PRINCIPLES OF DISTRIBUTED COMPUTING, 7., 1988. **Proceedings...** [S.l.:s.n.], 1988. p. 248-262.

NEKO. **The Neko Project**. Disponível em: <http://lsrwww.epfl.ch/neko>. Acesso em: jul. 2004.

NI, Q.; TURLETTI, T.; FU, W. Simulation-based Analysis of TCP Behavior over Hybrid Wireless & Wired Networks. In: INTERNATIONAL WORKSHOP ON WIRED/WIRELESS INTERNET COMMUNICATIONS, WWIC, 1., 2002. **Proceedings**… Las Vegas, Nevada: CSREA Press, 2002.

NS-2. The Network Simulator**.** Disponível em: <http://www.isi.edu/nsnam/ns>. Acesso em: set. 2005.

RODRIGUES, L. A. **Extensão do Modelo de Defeitos do Framework para Simulação de Algoritmos Distribuídos no Neko**. 2005. Trabalho Individual (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre, 2005.

RODRIGUES, L. A.;JANSCH-PÔRTO, I. Extensão do Suporte para Simulação de Defeitos no Neko. In: WORKSHOP ON THESES AND DISSERTATIONS IN DEPENDABLE COMPUTING, WTD, 3., 2005. **Proceedings**… [S.l.:s.n.], 2005.

SHANNON, R. E. **System Simulation:** The Art and Science. Englewood Cliffs, NJ: Prentice-Hall, 1975.

SSFNET. Network Simulator**.** Disponível em: <http://www.ssfnet.org/>. Acesso em: out. 2005.

TRINDADE, R. de M. **Uso do network simulator-NS para simulação de sistemas distribuídos em cenários com defeitos**. 2003. 133 f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

TRUMBO, B. E. **Session 2:** Two-State Markov Chains. Disponível em: <http://www.sci.csuhayward.edu/statistics/Gibbs/GibbsSess2.htm>. Acesso em: out. 2005.

URBÁN, P.; DÉFAGO, X.; KATAYAMA, T. NekoLS: prototyping and simulating Large Scale Distributed Systems. In: JOURNÉES SCIENTIFIQUES FRANCOPHONES, JSF, 2003, Tokyo, Japan. **Proceedings...** [S.l.: s.n.], 2003.

URBÁN, P.; DÉFAGO, X.; SCHIPER, A. Neko: A Single Environment to Simulate and Prototype Distributed Algorithms. In: IEEE INT'L CONF. ON INFORMATION NETWORKING, ICOIN, 15., 2001. **Proceedings…** [S.l.]:IEEE, 2001. p. 503-511.

VOIP. **Indepth: Packet Loss Burstiness**. Disponível em: <http://www.voiptroubleshooter.com/indepth/burstloss.html>. Acesso em: nov. 2005.

APÊNDICE A MANUAL DO NEKO

As próximas páginas referem-se ao documento que descreve as principais características do Neko, incluindo instalação, configuração e principais mecanismos, incluindo o suporte a defeitos desenvolvido neste trabalho. Inclui também detalhes sobre a ferramenta de visualização de *logs* da simulação, LogView, disponibilizada em conjunto com o pacote Neko. Ouras informações e exemplos também podem ser encontradas no capítulo 3 - O Framework Neko e no APÊNDICE B CD-ROM.

Para facilitar o acesso ao material, o manual foi escrito em inglês.

APÊNDICE B CD-ROM

O CD-ROM contém: a versão do Neko utilizada no desenvolvimento, os códigos-fonte escritos, os testes realizados com seus arquivos de configuração e resultados, o manual elaborado e o texto desta dissertação.