Arquitetura de um Framework de Injeção de Falhas Simuladas para Avaliação de Sistemas Distribuídos

Ruthiano S. Munaretti, Marinho P. Barcellos*

ruthiano@gmail.com, marinho@unisinos.br

Resumo

Simulação é uma importante ferramenta para avaliação de sistemas distribuídos, devido ao ambiente controlado que os mesmos oferecem. Injeção de falhas, por sua vez, é uma importante técnica para a avaliação da dependabilidade destes sistemas, através da identificação de erros em mecanismos de tolerância a falhas. Neste contexto, o presente relatório apresenta a arquitetura de um framework de injeção de falhas simuladas, baseada em um framework de simulação existente. Ênfase é dada ao funcionamento interno desta arquitetura, através da descrição dos componentes essenciais para a injeção de falhas. Ao final, o andamento do trabalho é analisado em função do projeto proposto e seu cronograma.

Palavras-Chave: Tolerância a Falhas, Simulação, Sistemas Distribuídos.

Abstract

Simulation is an important tool for distributed systems evaluation, because of its controlled environment. Fault injection, on its turn, is an important technique for dependability evaluation of these systems, through identification of errors in fault tolerance mechanisms. Thus, the present report presents the architecture of a simulated fault injection framework, based on an existed simulation framework. Emphasis is given to internal functioning of this architecture, through description of essential components to fault injection. At the end, the current work is analyzed in function of proposed project and the task list defined.

Keywords: Fault Tolerance, Simulation, Distributed Systems.

1 Introdução

Um sistema distribuído possui diversas características peculiares, envolvendo principalmente a sua parte estrutural, que pode afetar o funcionamento do sistema se não for devidamente considerada. Neste contexto é inserida a **avaliação** de sistemas distribuídos, onde a simulação é uma importante ferramenta, por fornecer um ambiente controlado para a realização da mesma. Certas aplicações distribuídas necessitam também de outras características, como confiabilidade e disponibilidade, devido aos serviços que as mesmas oferecem. Estas características podem ser mensuradas através da noção de *dependabilidade*, definida em [12] como a garantia de um serviço proporcionada por um sistema. Neste contexto, a injeção de falhas insere-se como uma importante técnica, uma vez que a mesma permite identificar a presença de erros em mecanismos de tolerância a falhas, bem como fornecer informações a respeito de sua eficiência [4].

Desta forma, a união da simulação com a injeção de falhas permite entender o comportamento dos sistemas distribuídos perante a existência de falhas. Nesse contexto, em [13] foi definida uma extensão para o Simmcast [11], um framework para simulação de protocolos de comuni-

*Orientador

cação e sistemas distribuídos. Neste trabalho, foi definido um conjunto de falhas, bem como o comportamento das mesmas perante os componentes do framework atual. Além disso, foram definidos também os mecanismos de ativação/desativação de falhas, para delinear o escopo das falhas nos componentes do simulador. Posteriormente, em [9], foi realizado o estudo e a definição do ciclo de vida de uma mensagem no simulador, juntamente com a criação de uma *matriz de contato*, utilizada para ativação de falhas no momento em que uma mensagem entrasse em contato com um determinado componente.

Ademais, ainda não existe uma arquitetura definida e, conseqüentemente, uma implementação desta extensão para o suporte a falhas no framework de simulação Simmcast. Assim, o presente relatório define a arquitetura do framework de injeção de falhas, abrangendo como cada um dos componentes definidos em [9, 13] insere-se no framework de simulação atual.

Assim, o artigo está organizado da seguinte maneira: a seção 2 apresenta o panorama atual da arquitetura do Simmcast, enquanto que a seção 3 aborda o funcionamento interno desta arquitetura, com ênfase nos componentes que serão estendidos. A seção 4 descreve a arquitetura estendida, com o suporte a falhas adicionado. Finalmente, a seção 5 tece considerações sobre o trabalho já realizado e os passos mais imediatos a seguir.

2 Arquitetura do Simmcast

A arquitetura do Simmcast, conforme [10], pode ser representada em **camadas**. Esta representação permite dividir os componentes por níveis de complexidade, onde os componentes de níveis inferiores oferecem serviços para os componentes de níveis superiores. As camadas que formam a arquitetura do Simmcast são ilustradas na figura 1.[1]

- **Java:** representada pela Máquina Virtual Java, esta camada oferece o suporte a *threads*, a unidade básica de execução do simulador.

- **Engine:** esta camada representa o motor de simulação, implementado como uma máquina de eventos discretos baseada em processos, sendo o processo correspondente a uma especialização da thread Java, retratada na camada anterior.

- **Kernel:** é a camada correspondente ao kernel do simulador, onde são definidos os componentes fundamentais para simulação de protocolos: **nodo, grupo, pacote, caminho** e **rede**.

- **Simulação:** camada responsável pela montagem de um *ambiente* de simulação, que é elaborado através da extensão dos componentes definidos na camanda **Kernel**. Esta camada é dividida em três grupos, descritos a seguir.

  - **Protocolo:** representa a *lógica* do protocolo a ser simulado. sendo um grupo de presença obrigatória.
  - **Roteamento:** grupo opcional, utilizado nas ocasiões em que o protocolo a ser simulado necessita de componentes pertencentes à estrutura física da rede.
  - **Aplicação:** grupo necessário nas situações em que se deseja verificar o comportamento de uma determinada aplicação perante o experimento de um determinado protocolo, sendo também opcional.

- **Configuração:** consiste de um mecanismo para criação de um *cenário* de simulação, baseado no ambiente definido e criado na camada anterior, sendo formado basicamente pela **topologia** em que o protocolo será simulado.

Por se tratar de um *framework*, o Simmcast possui os pontos de extensão definidos nas camadas **Kernel** e **Rotea-**

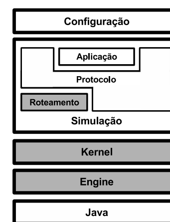

Figura 1: Camadas da arquitetura do Simmcast.

[1] A arquitetura em camadas descrita a seguir representa uma versão aperfeiçoada e substancialmente detalhada em relação à [10].

mento. Ainda na figura 1, as camadas destacadas na cor cinza são implementadas pelo Simmcast. As demais camadas, com exceção da camada Java, são implementadas pelo usuário do simulador.

3 Arquitetura Interna

A seção anterior apresentou uma visão *conceitual* da arquitetura do Simmcast, abrangendo o conteúdo de cada camada, bem como a interação entre elas. Segundo [9], estas camadas podem ser organizadas em duas partes distintas, retratadas a seguir.

- **Parte Superior:** representa o *sistema alvo* a ser simulado, envolvendo as camadas de **Configuração** e **Simulação**.

- **Parte Inferior:** representa o *sistema físico* no qual o sistema alvo é executado, envolvendo as camadas **Kernel** e **Engine**.

Para a realização da *extensão* desta arquitetura, é necessária a abordagem do funcionamento **interno** de cada uma das camadas envolvidas. Desta forma, as funcionalidades adicionais podem ser implementadas sem efeitos colaterais no funcionamento atual do simulador. Neste contexto, esta seção descreve a arquitetura interna das camadas **Kernel** e **Engine**. Serão abordadas somente estas duas camadas, uma vez que a injeção de falhas é aplicada na *Parte Inferior*, a fim de monitorar o comportamento da *Parte Superior* sob condições de falhas [9].

3.1 Engine

A camada Engine implementa o motor de simulação do framework, sendo responsável pelo *escalonamento de tarefas* a serem executadas pelo simulador. Neste modelo de escalonamento, somente um processo pode estar executando por vez. Em termos de funcionalidades, esta camada pode ser comparada ao escalonador de tarefas existente no sistema operacional Linux, por exemplo. Desta forma, assim como o escalonador de tarefas do Linux, as tarefas a serem realizadas na Engine estão armazenadas em **processos.** Entretanto, o modelo de escalonamento adotado nesta camada é **cooperativo**, diferentemente do modelo de escalonamento adotado no Linux, que é **preemptivo**. Isto se faz necessário para a garantia da *repetibilidade de experimentos*, uma vez que, em um modelo preemptivo, a ordem de execução das tarefas não é garantida.

Internamente, a Engine garante o funcionamento deste modelo de escalonamento através do uso de **monitores** (disponíveis na Máquina Virtual Java). Nestes monitores, a sincronização de threads é por *cooperação*, através dos métodos `wait()` e `notify()` da classe `Object`, sendo um objeto por processo. Junto com cada monitor é utilizado um **semáforo**, para garantir que o objeto em questão está sendo bloqueado/desbloqueado somente uma vez [14].

Quanto aos componentes, a camada Engine é formada basicamente por *processos*. Estes processos, por sua vez, são organizados dentro de uma *fila*. Ambos os componentes serão descritos com mais detalhes a seguir.

3.1.1 Processo

É o componente mais importante do motor de simulação, consistindo na sua unidade básica de execução. Na Engine, todo processo possui um *ciclo de vida*, podendo assim estar em cinco estados diferentes, ilustrados na figura 2 e descritos a seguir.

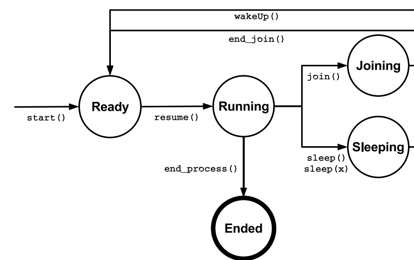

Figura 2: Diagrama de Estados de um Processo.

- **Ready:** o estado *Ready* representa um processo pronto para execução, ou seja, um processo que está esperando a vez de ser escalonado para execução.

- **Running:** representa um processo escalonado e em execução. Vale lembrar que, em um dado momento,

existe no **máximo** um processo neste estado durante um experimento.

- **Sleeping:** indica um processo cuja execução está suspensa. Em outras palavras, representa um processo que, voluntariamente, realizou o bloqueio de sua execução, a fim de aguardar a ocorrência de uma determinada situação no experimento.

- **Joining:** assim como o estado *Sleeping,* um processo em estado *Joining* também tem a sua execução bloqueada. Entretanto, a finalidade deste bloqueio é aguardar o término de algum outro processo.

- **Ended:** indica um processo cuja execução foi finalizada, ou seja, sua tarefa a ser realizada foi cumprida. Este estado é representado como um *estado final*, não conseguindo mais atingir os demais estados.

É importante ressaltar que um processo no estado *Running* **não** pode atingir os estados *Joining*, *Sleeping* e *Ended* simultaneamente. Isto ocorre porque a execução de cada uma das chamadas que levam à estes estados exige que o processo em questão esteja no estado *Running*, como será visto logo em seguida.

Cada processo possui uma *API*[2] *de utilização*, responsável pelas transições do processo nos diferentes estados possíveis. O uso desta API está ilustrada nas transições da figura 2 e descrito a seguir. Para um melhor entendimento de cada uma destas chamadas da API nos itens subseqüentes, é assumida a existência dos processos *p*, *q* e *r*, correspondente aos processos que *executam*, *sofrem* ou *executam e sofrem* a ação da chamada, respectivamente.

- **start():** responsável pela inicialização do processo *q*, esta chamada inclui efetivamente o processo no escalonador, colocando o mesmo no estado inicial *Ready*.

- **resume():** esta chamada realiza o escalonamento do processo *q*, situado no estado *Ready*, sendo executada pelo escalonador quando o processo *p*, situado no estado *Running*, deseja abandonar este estado.

- **join():** chamada executada por um processo *p*, situado no estado *Running*, quando o mesmo necessita de resultados que só podem ser obtidos após a realização de uma tarefa delegada a um processo *q*, situado no estado *Ready*.

- **sleep():** a chamada **sleep()**, executada por um processo *r* situado no estado *Running*, bloqueia a execução deste processo *r* por tempo indeterminado, passando-o assim para o estado *Sleeping*.

- **sleep(x):** esta chamada é análoga à chamada **sleep()** descrita anteriormente, com a diferença de que o bloqueio de execução é de um tempo **x** determinado.

- **wakeUp():** chamada realizada quando o processo *p,* situado no estado *Running*, deseja desbloquear a execução do processo *q*, situado no estado *Sleeping*.

- **end_process():** chamada que representa o término natural da execução do processo *r*. Após esta execução, o processo *r* passa para o estado final *Ended*.

- **end_join():** complementar à **join()**, esta chamada é executada pelo processo *p* quando a execução do mesmo está prestes a terminar e existe um outro processo *q* aguardando os seus resultados.

Além das chamadas acima, a API de processos também possui uma chamada denominada **run()**, responsável por armazenar a tarefa a ser executada pelo processo. Esta chamada é *abstrata* no contexto da Engine, sendo a implementação da mesma responsabilidade das camadas superiores. Durante esta execução, o estado do processo pode ser modificado para *Joining*, *Sleeping* ou *Ended*, através das chamadas **join()**, **sleep()**, **sleep(x)** ou **end_process()**, que foram explicadas anteriormente.

3.1.2 Fila de Processos

A fila de processos, no contexto da Engine, tem por objetivo organizar os processos existentes segundo uma determinada **ordem**. O seu funcionamento básico, baseado em [8], é realizado através dos componentes descritos a seguir.

- **Evento:** é um componente representado através de uma *tupla*, contendo o processo propriamente dito mais o tempo em que o mesmo deve ser escalonado pela Engine.

[2] Application Programming Interface

- **Fila Interna:** consiste em uma fila formada por eventos que possuem o *mesmo* tempo de escalonamento. Neste caso, a ordem dos eventos é definida pela **chegada** dos mesmos nesta respectiva fila.

Conseqüentemente, a Fila de Processos é formada por um conjunto de Filas Internas. A ordenação da Fila de Processos é realizada através do **tempo de escalonamento** da cabeça de cada uma das Filas Internas, começando do menor tempo. Um exemplo de Fila de Processos é ilustrado na figura 3.

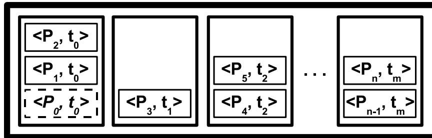

Figura 3: Exemplo de Fila de Processos.

Vale destacar ainda que o **próximo processo que será executado** pela Engine, bem como o **novo valor do relógio de simulação**, são representados pela *cabeça* da *primeira* Fila Interna. Na figura 3, este evento está destacado. Quanto à manipulação pela Engine, a Fila de Processos oferece duas operações, que serão descritas a seguir.

- **insertAt(t):** insere um processo na Fila de Processos, criando para isso um evento com o processo propriamente dito, mais o tempo de escalonamento **t**, sendo esta inserção realizada de forma **ordenada**.

- **removeFirst():** operação responsável pela remoção da *cabeça* da *primeira* Fila Interna, ou seja, do evento que contém o próximo processo que será executado pela Engine, bem como o novo valor do relógio de simulação, sendo utilizada para realizar o escalonamento de um novo processo.

Com relação à API de Processos (descrita na seção 3.1.1), algumas chamadas da mesma utilizam-se das operações da Fila de Processos. A seguir, são descritas as chamadas da API que fazem uso destas operações da Fila de Processos.

- **start():** inicializa o processo, inserindo-o na Fila de Processos para execução imediata, utilizando assim a operação **insertAt(t)**.

- **resume():** realiza o escalonamento para execução do processo. Logo, a chamada utiliza-se da operação **removeFirst()** para a realização deste escalonamento.

- **sleep(x):** após bloquear a execução do processo, esta chamada faz uso da operação **insertAt(t)** para escalonar novamente o processo em um dado tempo futuro.

- **wakeUp():** após desbloquear a execução de um dado processo, esta chamada insere o mesmo na Fila de Processos para execução imediata, utilizando-se para isso a operação **insertAt(t)**.

Em linhas gerais, a **consistência** foi a prioridade na implementação da Fila de Processos. Por este motivo, a mesma possui um *trade-off* conhecido, relacionado ao desempenho. Isto ocorre porque as operações de inserção/remoção na Fila de Processos, além de não possuírem otimizações, são realizadas como *seções críticas* (através do operador `synchronized`). Entretanto, os testes realizados até então com esta implementação apresentaram-se satisfatórios, não justificando assim uma otimização imediata.

3.2 Kernel

A camada Kernel implementa o *núcleo* do simulador, sendo responsável pela representação do **sistema físico** no qual um determinado protocolo irá ser simulado. Como já citado na seção 2, estes componentes são: **nodo**, **grupo, caminho, pacote** e **rede**. Em relação aos serviços, os principais referentes a esta camada estão relacionados ao **envio** e **recebimento** de pacotes, o principal meio de comunicação existente no sistema físico.

Nos itens subseqüentes, serão descritos cada um dos componentes essenciais da camada Kernel. Em seguida, será abordado o funcionamento interno do envio e recebimento de pacotes, representados pelas primitivas `Send` e `Receive`, respectivamente.

3.2.1 Nodo

O nodo é o principal componente da camada Kernel. Este componente é um dos *hot-spots* do framework de simulação, sendo assim um componente genérico. Como exemplo, um nodo pode representar um computador (no caso do sistema físico ser representado por uma rede de computadores) ou um roteador (no caso da camada Roteamento estar sendo utilizada).

A estrutura interna de um nodo engloba diversos componentes. Esta estrutura, com seus componentes principais, é ilustrada na figura 4 e descrita a seguir.

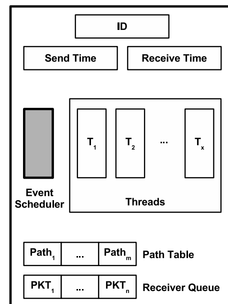

Figura 4: Estrutura interna de um nodo.

- **ID:** identificador numérico, responsável por identificar unicamente um nodo dentre um conjunto de nodos existentes em um determinado ambiente de simulação.
- **Send Time:** valor numérico, em unidades de simulação, que indica o tempo necessário para a transmissão de uma mensagem por uma determinada thread de um nodo.
- **Receive Time:** valor numérico, em unidades de simulação, indicando o tempo necessário para o recebimento de uma mensagem pela thread de um nodo.
- **Event Scheduler:** representa uma thread interna, implementada como um Processo da camada Engine, existente em todo e qualquer nodo.
- **Threads:** implementadas como um Processo da camada Engine (assim como o item anterior), as threads possuem a lógica de execução do nodo em questão.
- **Path Table:** representam os caminhos pelos quais o nodo está conectado com outros nodos no sistema físico modelado.
- **Receiver Queue:** este componente representa uma fila de pacotes, onde estão localizados os pacotes em recebimento no instante de tempo atual da simulação.

3.2.2 Grupo

O componente Grupo consiste em um conjunto de Nodos, que são diretamente conectados através de Caminhos. O objetivo da formação deste conjunto é inerente ao protocolo a ser simulado.

No contexto do Simmcast, este componente permite a simulação de protocolos de comunicação em grupo [5]. No contexto deste trabalho, uma atenção especial é voltada aos protocolos de comunicação em grupo tolerantes a falhas [1, 2], no qual este suporte a grupos oferece a possibilidade da realização de experimentos.

Quanto à sua estrutura interna, um Grupo possui os componentes e primitivas ilustrados na figura 5 e descritos a seguir.

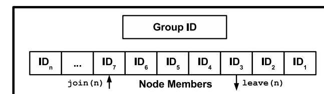

Figura 5: Estrutura interna de um grupo.

- **Group ID:** identificador numérico, responsável por identificar unicamente um grupo dentre um conjunto de grupos existentes no ambiente de simulação.
- **Node Members:** corresponde a um vetor contendo identificadores (IDs) de nodos, ou seja, os membros que fazem parte do grupo em questão.
- **`join(n)`:** primitiva, utilizada pelo usuário do simulador, para incluir o nodo **`n`** (especificado no parâmetro) em um determinado grupo.

- **`leave(n)`:** primitiva, utilizada pelo usuário do simulador, para retirar o nodo **`n`** (especificado no parâmetro) de um determinado grupo.

Outra particularidade refere-se ao **envio** de mensagens, procedente de um nodo com destino a um determinado grupo. Este assunto será abordado em mais detalhes na subseção 3.2.6.

3.2.3 Caminho

Um caminho consiste em um canal de comunicação unidirecional entre dois nodos quaisquer dentro do sistema físico modelado. Logo, toda e qualquer comunicação realizada entre nodos passa por um ou mais caminhos. Em relação à estrutura interna, um caminho possui os componentes ilustrados na figura 6, que são descritos a seguir.

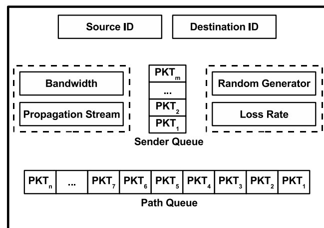

Figura 6: Estrutura interna de um caminho.

- **Source ID:** identificador do nodo *origem* do caminho. Neste caso, corresponde ao nodo que **transmite** pacotes para o caminho considerado.
- **Destination ID:** corresponde ao nodo *destino* do caminho, ou seja, ao nodo que **recebe** pacotes do caminho em questão.
- **Propagation Stream:** distribuição de dados, retornando um valor determinístico ou aleatório, responsável pela obtenção da *latência de propagação* de cada pacote no caminho.
- **Bandwidth:** corresponde à largura de banda do respectivo caminho, indicando a **capacidade** do mesmo.
- **Random Generator:** implementa um gerador de números randômicos convencional, que retorna valores em ponto flutuante dentro do intervalo $[0, 1]$, a fim de realizar a implementação da perda de pacotes no caminho, juntamente com o componente *Loss Rate*, descrito no próximo item.
- **Loss Rate:** valor em ponto flutuante, dentro do intervalo $[0, 1]$, que configura um **percentual de perda** de pacotes para um determinado caminho, com o objetivo de realizar a **perda de pacotes** no mesmo.
- **Sender Queue:** representa uma **fila de envio** de pacotes, abrangendo os pacotes em transmissão pelo nodo *origem* do caminho.
- **Path Queue:** fila que representa os pacotes que estão *efetivamente* no caminho, ou seja, os pacotes em propagação pelo mesmo.

3.2.4 Pacote

O pacote é a unidade básica de comunicação entre nodos no Simmcast. A criação de um pacote ocorre na thread de um nodo, no momento em que o mesmo deseja realizar o envio de dados para algum outro nodo existente no sistema físico modelado.

Com relação a estrutura interna, um pacote é formado pelos componentes ilustrados na figura 7 e descritos a seguir.

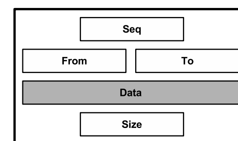

Figura 7: Estrutura interna de um pacote.

- **Seq:** número seqüencial, corresponde ao identificador numérico do pacote. Este número é gerado automaticamente pelo simulador.
- **From:** identificador numérico (ID) do nodo *origem*, ou seja, do nodo que criou e enviou o pacote.

- **To:** identificador numérico (ID) do nodo *destino*, ou seja, do nodo que receberá o pacote. Se o destino do pacote é um **grupo**, este campo é preenchido com o identificador numérico do respectivo grupo.

- **Data:** componente que representa o **conteúdo** do pacote. No contexto do Simmcast, este componente é **genérico**, ou seja, o formato deste conteúdo é dependente do protocolo a ser simulado.

- **Size:** valor numérico que indica o **tamanho** do pacote, não possuindo relação alguma com o tamanho da estrutura de dados relativa ao pacote no simulador. O valor é utilizado no experimento, como na verificação da propagação em um dado caminho, por exemplo.

3.2.5 Rede

Conceitualmente, o componente Rede representa o conjunto de Nodos, Grupos e respectivos Caminhos montados pelo usuário do simulador durante a modelagem do sistema. Assim como os componentes anteriores, este componente também é um *hot-spot* do framework, sendo estendido na criação do protocolo para a adição de funcionalidades específicas.

No contexto do Simmcast, este componente é implementado como um Processo da camada Engine, sendo responsável por toda a **inicialização** do experimento. Com relação a esta inicialização, os itens considerados são descritos a seguir.

- Leitura da camada de Configuração (para obter a topologia do sistema físico desejado).

- Definição da topologia, referente à camada Simulação.

- Criação dos Nodos (e instanciação de suas respectivas threads), Grupos e Caminhos da topologia acima, referente à camada Kernel.

- Inicialização do escalonador de processos, referente à camada Engine.

Como é possível observar acima, a abordagem de inicialização do componente Rede é *top-down*, começando da camada mais próxima ao usuário do simulador (onde o mesmo especifica o que deseja) e encerrando-se na camada mais baixa (Engine), onde o experimento é iniciado. Após realizar estas tarefas, a execução do processo correspondente ao componente Rede é suspensa até o fim do experimento.

3.2.6 Funcionamento da Primitiva `Send`

A primitiva `Send`, como já mencionada em seções anteriores, tem como objetivo principal realizar o **envio** de informações, procedentes de um determinado nodo, para um outro nodo ou grupo. O estudo de seu funcionamento interno permite identificar a forma pela qual a comunicação é realizada entre os diversos nodos de um ambiente de simulação. Para um melhor entendimento, o funcionamento interno da primitiva `Send` é dividido em duas fases, abordadas logo em seguida.

A primeira fase, denominada como **fase 1**, consiste na *preparação* das informações a serem enviadas de um nodo origem para um destinatário (nodo ou grupo). Esta fase é composta por três procedimentos, identificados numericamente na figura 8, sendo os procedimentos realizados nesta ordem especificada pelos rótulos numéricos. As tarefas executadas em cada um destes procedimentos são descritas a seguir.

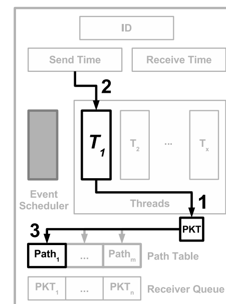

Figura 8: Fase 1 do envio de um pacote.

1. Criação de um **pacote** pela thread do nodo que requisitar a primitiva `Send`.

2. Bloqueio de execução da respectiva thread que requisitou a criação do pacote, a fim de realizar a simulação do *processamento* de um pacote.

3. Inclusão do pacote criado na **fila de envio** do nodo.

A próxima fase, denominada **fase 2**, abrange o *envio* propriamente dito do pacote preparado pelo nodo remetente. Esta fase é composta de cinco procedimentos, identificados numericamente na figura 9, sendo os mesmos realizados nesta ordem apresentada. As tarefas de cada um destes procedimentos são abordadas a seguir.

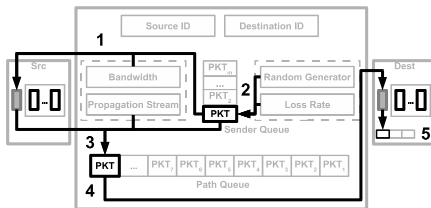

Figura 9: Fase 2 do envio de um pacote.

1. Criação de um **evento de partida** no *Event Scheduler* do nodo remetente, consistindo em um tempo futuro (calculado através do atributo **Bandwidth**) no qual o pacote será retirado da cabeça da fila de envio e colocado na fila do caminho.

2. Verificação de **perda do pacote**, através dos atributos *Random Generator* e *Loss Rate*. Assim, o pacote pode ser descartado ou não, dependendo do resultado deste algoritmo.

3. Se o pacote não for descartado no algoritmo de perda, o mesmo passa a estar em propagação pelo caminho, permanecendo assim na **fila do caminho**.

4. Com o pacote presente na fila do caminho, um **evento de chegada** é criado no *Event Scheduler* do nodo destino, cujo tempo é calculado através do atributo **Propagation Stream**.

5. Quando o evento de chegada criado for escalonado, o seu respectivo pacote será incluído na **fila de recebimento** do nodo destino, estando assim o pacote pronto para ser recebido pelo nodo.

Desta forma, a execução da primitiva `Send` é bem sucedida quando o pacote criado atinge a fila de recebimento do nodo destino. Em contrapartida, a execução desta primitiva não é bem sucedida quando o respectivo pacote é descartado em uma das filas percorridas pelo mesmo, em razão de uma possível capacidade esgotada. Na próxima seção, será abordado o funcionamento da primitiva `Receive`, que complementa a abordagem de comunicação entre nodos utilizada pelo simulador.

3.2.7 Funcionamento da Primitiva `Receive`

A primitiva `Receive` consiste em uma solicitação explícita para recebimento de pacote. Conseqüemente, a mesma é utilizada nas ocasiões onde um determinado nodo destino deseja receber um pacote, proveniente de um nodo origem.

Ao executar `Receive`, a tarefa da respectiva thread consiste basicamente na verificação da **fila de recebimento** de seu próprio nodo. Sobre esta fila, a mesma está implementada em um formato denominado de *multifila*, através de uma lista duplamente encadeada. Nesta lista, cada elemento possui quatro ponteiros: dois deles apontando para os elementos anterior e próximo da fila de pacotes recebidos de um determinado nodo origem, e os outros dois ponteiros apontando para os elementos anterior e próximo da fila **geral**, formada pelos pacotes recebidos de todo e qualquer nodo. Um exemplo de elemento existente na fila de recebimento é ilustrado na figura 10, com suas respectivas posições nas filas geral e específica.

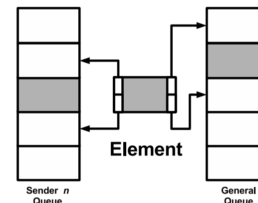

Figura 10: Elemento inserido na fila de recebimento.

Pelo tipo de uso, a primitiva `Receive` é dividida em três modalidades. Ambas são explicadas a seguir.

- **tryReceive:** como o próprio nome já indica, esta modalidade consiste na **tentativa** de recebimento de um determinado pacote. Por este motivo, a mesma é *não-bloqueante*, ou seja, não existe uma espera para o recebimento de pacotes.

- **receive:** modalidade que baseia-se no recebimento *bloqueante* de um determinado pacote, ao contrário da modalidade descrita no item anterior. Assim, a thread que solicitar a execução de **receive** *irá* aguardar a existência de pacotes na fila de recebimento, caso não existam pacotes no momento da execução.
- **receive(t):** modalidade análoga à descrita no item anterior, com a diferença de que o bloqueio de execução é de um tempo **t** previamente determinado.

Pela forma de implementação adotada na fila de recebimento, cada uma destas modalidades pode ser executada de duas formas diferentes. Ambas são descritas a seguir, usando-se a modalidade **tryReceive** como exemplo.

- **tryReceive():** executa o recebimento de um pacote proveniente de *qualquer* nodo origem. Neste caso, na fila de recebimento, são utilizados os ponteiros referentes à **fila geral**.
- **tryReceive(n):** recebe um pacote proveniente de um determinado nodo origem, especificado pelo parâmetro **n**. Neste caso, com relação à fila de recebimento, são utilizados os ponteiros referentes à **fila específica** de pacotes provenientes do nodo origem **n**.

Assim, a execução da primitiva `Receive` é bem sucedida quando um pacote é obtido com sucesso da fila de recebimento. Em contrapartida, a execução desta primitiva não é bem sucedida quando ocorre o contrário, ou seja, quando não se obtém pacote algum durante o tempo desta execução. Quando isto ocorre, é retornado um pacote com o valor `null`. No caso específico da modalidade **receive** bloqueante por tempo indeterminado, a thread que executou esta chamada pode permanecer bloqueada por todo o tempo restante do experimento. Isto ocorre porque a responsabilidade de desbloqueio da mesma é inerente ao protocolo a ser simulado, que deve ter o cuidado de sempre enviar pacotes para threads que executem esse tipo de chamada.

4 Arquitetura Estendida

A seção atual tem por objetivo delinear uma *extensão* para a arquitetura atual, com o objetivo de adicionar o suporte a falhas. Este suporte é realizado basicamente através do mecanismo de *injeção de falhas*, onde cada tipo de falha é adicionado à um determinado componente. Conseqüentemente, o componente em questão estará *sujeito* à ocorrência da respectiva falha adicionada.

Neste contexto, em [9] foram especificados o **modelo de falhas** adotado e o **mecanismo de ativação/desativação de falhas** utilizado. Estes itens são retomados nas subseções seguintes, com o intuito de realizar o mapeamento dos mesmos na arquitetura interna existente, formando assim a arquitetura estendida proposta.

4.1 Modelo de Falhas

Conforme descrito no início da seção 3, no contexto da arquitetura interna, a injeção de falhas será aplicada na *Parte Inferior* desta arquitetura, formada pelas camadas **Kernel** e **Engine**. Os componentes sujeitos a falhas estão presentes na camada **Kernel**, pois esta é a camada de mais baixo nível no contexto do simulador, oferecendo os componentes básicos para a construção de experimentos. Embora sofra alterações nos componentes utilizados pela camada **Kernel**, a camada **Engine** não é considerada no contexto de injeção de falhas em componentes, uma vez que a mesma é independente do simulador em si, oferecendo apenas suporte para escalonamento de tarefas em relação à sua camada imediatamente superior.

Como já citado na seção 2, os componentes da camada **Kernel** são: **nodo**, **grupo, caminho, pacote** e **rede**. Entretanto, a injeção de falhas é prevista apenas nos componentes **nodo**, **grupo** e **caminho**, conforme justificado em [9]. No contexto do simulador, a injeção de falhas é realizada nestes componentes através do mecanismo de *extensão*, a fim de manter a compatibilidade de funcionamento em experimentos com e sem uso de falhas.

Neste contexto, cada um destes componentes admite um determinado conjunto de falhas, conforme sua especificação, em termos de funcionalidade e de premissas sobre o ambiente [13]. Este conjunto, denominado como **modelo de falhas**, define categorias de falhas (tal como [3, 6, 7, 12]). Desta forma, a especificação de um sistema tolerante a falhas pode então se basear na nomenclatura apresentada por este modelo, visando definir precisamente quais condições são tratadas, o que assume-se não ocorrer, e o que não é tratado [13]. Em [9], foi definido um modelo de falhas, derivado do modelo proposto por

Veríssimo e Rodrigues em [12]. Este modelo, ilustrado na tabela 1, foi escolhido por possuir um nível adequado de abstração e ser orientado a sistemas distribuídos, um dos focos do framework de simulação.

Tipo	**Descrição**
Colapso	Componente pára silenciosamente de funcionar.
Omissão	Componente omite resultados, de forma completa ou parcial.
Temporização	Funcionamento com tempo arbitrário.
Sintática	Comportamento incorreto, detectável.
Semântica	Comportamento correto com sentido incorreto.

Tabela 1: Modelo de falhas.

Desta forma, cada tipo de falha provoca um determinado **comportamento** no respectivo componente sujeito a falhas. Estes comportamentos foram definidos em [9] e serão retomados na seções seguintes, classificados por tipo de falha e por componente. Além disso, será tratado também o mapeamento do respectivo comportamento na arquitetura interna, formando a arquitetura estendida proposta.

4.2 Falha de Colapso

A falha de colapso em um componente consiste na *interrupção* de seu processamento. Nesta interrupção, todas as informações de estado são perdidas, representando assim um *colapso com amnésia*. A *recuperação* de um componente em colapso no decorrer do experimento é prevista. Entretanto, como o colapso é com amnésia, as informações existentes anteriormente neste componente **não** são recuperadas, voltando-se assim aos valores iniciais disponibilizados no início do experimento.

Vislumbrando-se a arquitetura interna, a falha de colapso pode ser implantada no componente em questão através da criação de três **dispositivos** adicionais: um **limpador**, um **bloqueador** e um **desbloqueador**, responsáveis pela remoção de todo o conteúdo interno, pelo bloqueio e pelo desbloqueio de acesso. O funcionamento destes dispositivos ocorre da seguinte maneira: na *ativação* da falha, é aplicado o **limpador**; em seguida, entra em funcionamento o **bloqueador**, que permanece aplicado durante todo o tempo de ocorrência da falha; no *encerramento* da mesma, é aplicado o **desbloqueador**, devolvendo assim o acesso normal ao respectivo componente. Nas subseções a seguir, são descritas as implantações da falha de colapso para cada um dos componentes sujeitos a falhas, especificados na subseção 4.1.

Com relação ao componente nodo, o bloqueador é aplicado através da **interrupção de execução** de todas as threads do mesmo (inclusive da thread responsável pelo escalonamento de eventos). Para isso, é adicionado o estado *Stopped* ao diagrama de estados do processo. Neste estado, a execução do processo é interrompida e todas as suas informações de estado (como valores de variáveis internas, por exemplo) são reinicializadas. Para atingir o estado *Stopped*, a chamada **stop**() foi incluída na API de processos, juntamente com a chamada **end_stop()**, sendo esta a responsável pela recuperação do respectivo processo.

Além das threads, a falha de colapso em nodo afeta também a **fila de envio** (pois a mesma pertence *conceitualmente* ao nodo), além da **fila de recebimento**. Desta forma, ao ocorrer um colapso, o conteúdo de ambas as filas deve ser **perdido**, devido à amnésia de estado. Durante o colapso, o **acesso** à estas filas deve ser **bloqueado**, tanto para leitura quanto para gravação de dados, uma vez que o funcionamento do nodo está interrompido pela falha. Naturalmente, ao término da ocorrência da falha, o acesso às filas deve ser **desbloqueado**, devido à volta de funcionamento do nodo. Referente à arquitetura interna, os dispositivos para implantação do colapso (limpador, bloqueador e desbloqueador) são utilizados nas filas de envio e recebimento do respectivo nodo afetado, bem como em todas as threads do mesmo.

Com relação ao caminho, uma falha de colapso faz com que o funcionamento do mesmo seja interrompido. Como o colapso é com amnésia, todos os pacotes em propagação no mesmo são perdidos. Ao recuperar-se de um colapso, um caminho passa a funcionar normalmente, a partir do mesmo estado em que se situava no início do experimento. Além dos pacotes em propagação, a falha de colapso no caminho afeta também os demais componentes existentes, com exceção da **fila de envio**, uma vez que a mesma pertence conceitualmente ao nodo origem.

Em um grupo, a falha de colapso ocasiona o descarte de todos os pacotes destinados ao mesmo, sendo mantida a transmissão dos demais pacotes. Na arquitetura interna, os dispositivos para implantação do colapso atuam nos *membros* do respectivo grupo afetado, mais especifi-

camente nas filas de envio dos mesmos. Nestas filas, são descartados todos os pacotes cujo destinatário é o grupo afetado pela falha. Este descarte é mantido durante toda a ocorrência da falha, sendo suspenso na sua desativação.

4.3 Falha de Omissão

A falha de omissão em um determinado componente baseia-se essencialmente na não realização de um ou mais serviços fornecidos pelo mesmo. A mesma pode ser *total* ou *parcial*, delimitando assim o nível de omissão desejado, o que depende das necessidades do experimento. No contexto do simulador, as falhas de omissão afetam o envio de **pacotes**, uma vez que este é o principal serviço oferecido.

Desta forma, com relação à arquitetura interna, a falha de omissão é implantada nos componentes afetados através de um dispositivo denominado **filtro de pacotes**. Seu funcionamento ocorre de maneira análoga à perda de pacotes pelo caminho, descrita na subseção 3.2.3. Primeiramente, um novo valor randômico é sorteado de um gerador, utilizado especificamente para este propósito. Logo após, este valor sorteado é comparado com o **percentual de omissão**, um valor no intervalo $[0, 1]$ denominado de *Omission Rate*, previamente configurado pelo usuário na descrição do experimento. Se o valor sorteado for menor que o percentual de omissão configurado, o pacote em questão é omitido. Caso contrário, o pacote em questão é mantido. Nas subseções seguintes, são descritos os comportamentos dos componentes afetados perante a aplicação da falha de omissão.

No contexto do nodo, a aplicação de uma falha de omissão no mesmo **não** afeta o processamento, o recebimento e o envio de pacotes. Neste caso, o serviço afetado é a chegada, ao respectivo caminho, dos pacotes que estão sendo enviados. Assim, para o receptor, a disponibilização de um pacote no caminho estará sendo omitida. Um nodo sofre uma falha de *omissão total* se, ao realizar o envio de pacotes, a disponibilização destes pacotes no caminho não for efetivada. A falha de *omissão parcial*, por sua vez, ocorre quando apenas alguns pacotes são disponibilizados no caminho, enquanto os pacotes restantes são descartados. Na arquitetura interna, o dispositivo **filtro de pacotes** é aplicado ao pacote situado na cabeça da fila de envio, relembrando que esta fila é situada conceitualmente no nodo. Esta aplicação é realizada no momento imediatamente anterior ao início da fase 2 do envio de um pacote. Conseqüentemente, a fase 1 do envio não é afetada.

A falha de omissão aplicada em um caminho afeta os pacotes **em propagação** pelo mesmo. Estes pacotes estão localizados na **Path Queue**, descrita na subseção 3.2.3. Desta forma, um **filtro de pacotes** é instalado ao final desta fila, a fim de verificar os pacotes que acabaram de sair da fila de envio do nodo origem e, conseqüentemente, estão chegando no respectivo caminho.

Em grupos, uma falha de omissão realize o descarte dos pacotes em propagação, de forma *total* ou *parcial*, cujo destinatário é o respectivo grupo afetado. Estes pacotes estão localizados nas **filas de envio** dos *membros* deste grupo. Assim, é instalado um **filtro de pacote** no grupo, que atua diretamente sobre estas respectivas filas de envio.

4.4 Falha de Temporização

Falhas temporais em um dado componente fazem com que os serviços prestados pelo mesmo sejam executados de forma *atrasada* ou *adiantada*, em relação ao tempo existente no relógio de simulação. No contexto específico do simulador, uma falha temporal afeta predominantemente o **envio de pacotes**, que consiste no principal serviço oferecido pelo mesmo. Desta forma, será descrito a seguir o comportamento de cada componente sujeito a falhas, perante a falha de temporização.

Em relação à arquitetura interna, é importante ressaltar que a mesma possui apenas um relógio de simulação. Este relógio, presente na camada **Engine**, é denominado de **relógio global**, uma vez que o mesmo é responsável pela geração de todos os tempos utilizados pelo simulador. Logo, não existe na arquitetura interna atual a noção de **relógios locais**, necessários para a implementação de falhas de temporização em nodos. Neste contexto, será adicionado na arquitetura interna atual um **relógio local** para cada nodo existente no experimento. Este relógio local estará associado ao escalonador de eventos do respectivo nodo. Logo, o mesmo será responsável pelos tempos de envio e recebimento de mensagens. A atualização do relógio local ocorrerá no mesmo momento da atualização do relógio global, por este motivo, ambos os relógios são **sincronizados** no decorrer de um experimento.

No componente nodo, além do envio de pacotes, a ocorrência de uma falha temporal afeta também os demais serviços oferecidos por este componente. Assim, todos estes

serviços podem estar executando de forma *atrasada* ou *adiantada* em relação ao relógio de simulação, o que depende do cenário desejado pelo usuário. Assim, é adicionado neste componente uma variável numérica denominada ***drift***, cujo objetivo é alterar a velocidade de um determinado relógio local. Desta forma, o nodo passará a executar os seus respectivos serviços de forma *atrasada* ou *adiantada*, comparando-se com o tempo existente no relógio global, que não sofre o efeito do *drift* em hipótese alguma. O valor de *drift* é aplicado a um determinado relógio local no momento da atualização de todos os relógios da simulação (global e locais), através da fórmula apresentada a seguir, onde T_1 é o valor atual do relógio local em questão e x é a quantidade de tempo relativo a ser adicionada nos relógios de simulação durante a atualização dos mesmos.

$$T_1 = T_1 + (x \star drift)$$

Falhas temporais em caminhos ocasionam *atrasos* na transmissão de dados, porém nunca *adiantamentos*, devido ao limite de banda que os mesmos possuem. Desta forma, esta falha afeta os pacotes em propagação, através da aplicação de um **delay** para *todos* os pacotes que chegam no caminho (ou seja, que acabaram de sair da fila de envio).

Na aplicação da falha de temporização em um determinado grupo, os pacotes com destino ao mesmo sofrem *atrasos* na transmissão. *Adiantamentos* não são previstos, devido aos limites de banda inerentes aos caminhos que interligam os membros do respectivo grupo. Esta falha é aplicada nos pacotes das **filas de envio** de todos os membros pertencentes ao grupo, afetando somente os pacotes cujo destinatário seja o respectivo grupo afetado. A aplicação da falha, por sua vez, consiste na adição de um **delay**, responsável por ocasionar o atraso desejado aos pacotes.

4.5 Falha Sintática

Uma falha sintática em um componente consiste no **comportamento incorreto** do mesmo que, no entanto, pode ser detectado por outros componentes que estejam livres de falhas. A atuação desta falha ocorre nos campos **From**, **To** e **Data** do pacote afetado. Neste caso, os identificadores de nodo dos campos **From** e **To** são trocados por identificadores inexistentes no experimento, afetando assim o envio ao destinatário (no caso de troca no campo **To**) e o envio de uma confirmação de recebimento (no caso de troca no campo **From**). Além disso, o *tipo* de dados existente no campo **Data** é trocado por um tipo desconhecido pelo experimento, bem como o próprio conteúdo deste campo (o que é dependente do experimento), corrompendo assim o conteúdo do pacote. Vale ressaltar que este tipo de falha pode afetar um ou mais destes três campos do pacote, sendo esta escolha dos campos afetados inerente ao cenário de simulação desejado pelo usuário.

Em relação à arquitetura interna, é adicionado um **filtro de pacotes** para a implementação de uma falha sintática. Além disso, é configurado também um **percentual** de pacotes afetados, denominado de **Syntactic Rate**, juntamente com um gerador de números randômicos. O funcionamento deste percentual de pacotes afetados é análogo ao descrito na subseção 3.2.3. Assim, são explicados nas subseções a seguir os comportamentos de cada um dos componentes sujeitos a falhas perante a falha sintática.

No contexto do componente nodo, esta falha é aplicada no **envio de pacotes**, o principal serviço oferecido pelo mesmo. Assim, a aplicação da falha sintática no nodo faz com que o mesmo passe a enviar pacotes sintaticamente incorretos. Com relação ao filtro de pacotes, o mesmo é aplicado logo após a criação de um pacote por uma determinada thread de um nodo, atuando assim nos campos **From**, **To** e **Data** do respectivo pacote. Assim, a fase 1 do envio de pacotes é afetada.

A falha sintática em um caminho é aplicada na fase de propagação, gerando assim pacotes sintaticamente incorretos. O filtro de pacotes, responsável pela aplicação da falha, é utilizado no final da **Path Queue**, ou seja, sobre os pacotes que acabaram de atingir esta respectiva fila.

No componente grupo, a falha sintática ocasiona a transmissão de pacotes sintaticamente incorretos. Neste contexto, o filtro de pacotes é instalado nas **filas de envio** de todos os membros de um determinado grupo, porém afetando somente os pacotes com o destinatário configurado para o respectivo grupo.

4.6 Falha Semântica

Com relação ao funcionamento geral, a falha semântica é semelhante à falha sintática. Entretanto, a falha semântica aplicada a um determinado componente não afeta o

comportamento do mesmo, mas sim a **semântica** deste comportamento. Logo, esta falha pode não ser detectável, uma vez que o comportamento do componente falho apresenta-se correto.

Assim como na falha sintática, a falha semântica atua-se nos campos **From**, **To** e **Data** do respectivo pacote afetado. Neste caso, os identificadores dos campos **From** e **To** são trocados por identificadores de *outros* nodos existentes no experimento; com relação ao campo **Data**, o tipo de dados do mesmo é trocado por um outro tipo existente no experimento, bem como o próprio conteúdo deste campo (o que é dependente do experimento que está sendo realizado), ambos não compatíveis com os dados existentes. Desta forma, o pacote afetado ainda é um pacote válido, entretanto, seu comportamento será incorreto: o mesmo será enviado para um nodo (ou grupo) diferente do especificado, a confirmação de recebimento será retornada para um nodo diferente do que enviou o respectivo pacote e, finalmente, os dados de um pacote não poderão ser recuperados, uma vez que o tipo especificado será incompatível com os mesmos.

Para a implementação da falha semântica na arquitetura interna, é instalado um **filtro de pacotes**. Junto com este filtro, é instalado um **percentual** de pacotes afetados, denominado de **Semantic Rate**, e um gerador de números randômicos. O funcionamento de todos estes componentes é análogo ao já descrito na falha sintática. Entretanto, é importante ressaltar que as falhas sintática e semântica são *mutuamente excludentes*, ou seja, não é possível aplicar ambas as falhas simultaneamente em um determinado componente. Neste contexto, são descritos a seguir os comportamentos de cada componente sujeito a falhas perante a aplicação da falha semântica.

A falha semântica em um nodo afeta o envio de pacotes do mesmo. Desta forma, após a ativação desta falha, o mesmo passa a enviar pacotes sintaticamente corretos, porém com semântica incorreta. Da mesma forma que a falha sintática, o filtro de pacotes é aplicado logo após a criação de um pacote por uma determinada thread de um nodo, atuando assim nos campos **From**, **To** e **Data** do respectivo pacote.

A falha semântica em um caminho é aplicada na fase de propagação, gerando assim os pacotes que possuem uma semântica incorreta. O filtro de pacotes, responsável pela aplicação da falha, é utilizado no final da **Path Queue**, ou seja, sobre os pacotes que acabaram de atingir esta respectiva fila.

No componente grupo, a falha semântica ocasiona a transmissão de pacotes que possuem uma semântica incorreta. Neste contexto, o filtro de pacotes é instalado nas **filas de envio** de todos os membros de um determinado grupo, porém afetando somente os pacotes com o destinatário configurado para o respectivo grupo.

4.7 Interface de Falhas

As subseções anteriores apresentaram os comportamentos de falhas, previstos para a arquitetura estendida do Simmcast, que atendem ao modelo de falhas adotado. Para efeito de definição, estes comportamentos podem ser agrupados em uma **camada interna**, uma vez que são parte integrante do **Kernel** do simulador. Logo, uma **interface de falhas** se faz necessária, a fim de que o usuário do simulador possa fazer uso adequado dos comportamentos definidos, incrementando assim seus experimentos.

Neste sentido, é definida uma **camada externa**, que consiste basicamente de uma interface para o uso dos comportamentos definidos. Esta interface é construída com base nas regras de ativação e desativação de falhas, retratadas em [9] e retomadas na subseção a seguir. Assim, esta interface externa define um **conjunto de falhas**, que podem ser aplicadas à um ou mais componentes pelo usuário do experimento.

Desta forma, o objetivo principal da interface externa é *delimitar* o escopo de falhas disponíveis em um dado experimento, adotando-se assim somente os comportamentos relevantes ao mesmo. Além disso, a interface externa permite também a *composição de comportamentos*, de forma que um determinado tipo de falha seja constituído de vários comportamentos. Assim, esta interface fornece mecanismos suficientes para que o usuário do simulador construa um conjunto de falhas adequado ao seu experimento.

Com relação à arquitetura interna do simulador, a interface externa é implantada através de uma *micro-linguagem*, que será posicionada na camada **Configuração**. Esta micro-linguagem consiste-se basicamente de um ou mais **nomes de falha**, onde cada nome de falha está diretamente associado com um ou mais **comportamentos.** A definição precisa desta micro-linguagem está em andamento e, no presente momento, um exemplo em

pseudolinguagem que ilustra o modelo de falhas visto na subseção 4.1 pode ser visualizado na figura 11.

```
Crash:     {Cleaner, Blocker, Unblocker}
Omission:  {OmissionFilter(Rate)}
Timing:    {Drift}
Syntactic:{SyntacticFilter(From, To, Data)}
Semantic:  {SemanticFilter(From, To, Data)}
```

Figura 11: Exemplo de interface externa.

4.8 Ativação/Desativação de Falhas

A partir da interface externa, descrita na subseção anterior, o usuário do simulador já possui o conjunto de falhas necessárias para o seu experimento. Entretanto, é necessário também um mecanismo para especificar *quando*, *como* e/ou *de que forma* estas falhas irão ocorrer durante o experimento. Para isso, é definido em [9] um mecanismo de **ativação/desativação de falhas**, que é retomado neste contexto, a fim de preencher esta lacuna existente até então na arquitetura estendida.

Neste mecanismo, existem duas modalidades de regras: a *regra de ativação* e a *regra de desativação*, utilizadas para **ativar** e **desativar** uma falha, respectivamente. Estas regras, por sua vez, podem ser de vários tipos, sendo a escolha desse tipo inerente ao cenário de simulação desejado. Estes tipos possíveis de regras são descritos nos itens a seguir.

- **Intervalos:** valor numérico, representando o tempo relativo em que, a partir de uma recuperação, uma determinada falha em um componente deve ser ativada. Além disso, este valor pode representar também o tempo relativo em que, a partir de uma falha, um determinado componente deve ser recuperado.

- **Expressões booleanas:** consistem-se de expressões que, no momento em que forem *verdadeiras*, ativam ou desativam um determinada falha. É importante ressaltar que expressões booleanas são compostas de *termos*. Estes termos, por sua vez, são separados por *operadores binários*, que podem ser **lógicos** (“&&” e “||”) ou **relacionais** (“==”, “>”, “<”, “>=”, “<=” e “!=”). No contexto do mecanismo definido, um termo pode ser um dos seguintes itens: referência ao relógio de simulação, conteúdo de pacotes, variáveis de distribuição aleatória ou estado interno de nodos.

Na arquitetura interna, este mecanismo é implantado na camada **Engine**, devido ao uso intensivo do relógio de simulação para a verificação de regras. Assim, é utilizado para isso um **Avaliador de Regras**, sendo responsável pela avaliação de todas as regras utilizadas pelo experimento segundo um *intervalo* de tempo pré-definido. Sobre este intervalo, é importante ressaltar um *trade-off* conhecido: quanto **menor** este valor, mais precisa será a avaliação de regras, o que impacta negativamente no desempenho; já valores **maiores** beneficiam o desempenho, mas com uma avaliação menos precisa de regras.

5 Conclusões

O relatório apresentou a arquitetura de um framework de injeção de falhas simuladas, baseada em um framework de simulação existente. O foco inicial foi na realização de uma descrição detalhada da arquitetura do framework existente, aperfeiçoando-se a descrição da mesma já existente em trabalhos anteriores. Logo após, foi realizado um estudo minucioso da arquitetura interna do framework de simulação atual, com ênfase no funcionamento interno dos principais componentes. Em seguida, foi abordada a arquitetura estendida, através da descrição dos comportamentos previstos nos componentes sujeitos a falhas do simulador, através de um modelo de falhas. Finalmente, foi descrita uma interface externa para a utilização destes comportamentos, juntamente com um mecanismo de ativação/desativação de falhas.

Com relação ao cronograma proposto, alguns pontos merecem ser destacados. O primeiro deles diz respeito aos itens referentes ao estudo da arquitetura interna, à identificação dos pontos de extensão e à arquitetura do modelo de falhas: ambos os itens foram realizados de forma completa, com os resultados presentes neste relatório. Com relação à fusão das arquiteturas, outro item previsto, a mesma foi sendo realizada gradativamente durante a execução de cada um dos itens anteriores, uma vez que a arquitetura estendida criada mostrou-se fortemente acoplada à arquitetura existente, graças ao mecanismo de extensibilidade já existente na mesma.

Referente à arquitetura de ativação/desativação de fa-

lhas, embora tenha sido bem desenvolvida, a mesma não foi completamente concluída até o momento da finalização deste relatório. Neste caso, certos pontos sobre a arquitetura interna ainda não foram especificados detalhadamente. Em compensação, o estudo do código-fonte do Simmcast e a elaboração da monografia foram adiantados neste período de trabalho. Desta forma, um tempo maior poderá será disponibilizado para a finalização da arquitetura de ativação/desativação de falhas, bem como para a realização dos experimentos de validação, outro item considerado fundamental para o trabalho.

Assim, a continuidade do trabalho será focada na finalização da arquitetura de ativação/desativação de falhas. Em seguida, será abordada a implementação de toda esta arquitetura estendida proposta. Após esta implementação, experimentos de validação serão aplicados na mesma, a fim de verificar todas estas funcionalidades adicionadas.

Referências

[1] Y. Amir and J. Stanton. The spread wide area group communication system. Technical Report CNDS 98-4, 1998.

[2] C. Basile, L. Wang, Z. Kalbarczyk, and R.K. Iyer. Group communication protocols under errors. In *22nd Symposium on Reliable Distributed Systems*, 2003.

[3] K. Birman. *Building Secure and Reliable Network Applications*. Prentice Hall, 1996. 500 p.

[4] A.R. Ejlali, S.G. Miremadi, H.R. Zarandi, G. Asadi, and S.B Sarmadi. A Hybrid Fault Injection Approach Based on Simulation and Emulation Cooperation. In *Proceedings of the IEEE/IFIP Internation Conference on Dependable Systems and Networks (DSN-2003)*, pages 479–488, 2003.

[5] H. H. Muhammad and M. P. Barcellos. Simulation group communication protocols through an object-oriented framework. In SCS, editor, *35th Annual Simulation Symposium, ANSS 2001*, volume 1, San Diego, USA, April 2001. SCS.

[6] V. Hadzilacos and S. Toueg. *Distributed Systems*, chapter 5, Fault-Tolerant Broadcasts and Related Problems, pages 97–146. Addison-Wesley, 2nd. edition, 1998.

[7] P. Jalote. *Fault Tolerance in Distributed Systems*. Prentice-Hall, 1998.

[8] M. C. Little. Construction and Use of a Simulation Package in C++. Technical Report 437, Department of Computing Science, University of Newcastle upon Tyne, 1993.

[9] M. P. Barcellos, C. Woszezenki, and R. Munaretti. Framework de Injeção de Falhas Simulada para Avaliação de Sistemas Distribuídos. In SBC, editor, *XXIII Simpósio Brasileiro de Redes de Computadores, SBRC 2005*, volume 1, Fortaleza, Brasil, maio 2005. SBC.

[10] M. P. Barcellos, G. Facchini, L. F. Cintra, and H. H. Muhammad. Projeto do Framework de Simulação Simmcast: uma Arquitetura em Camadas com Ênfase na Extensibilidade. In SBC, editor, *XXII Simpósio Brasileiro de Redes de Computadores, SBRC 2004*, volume 1, Gramado, Brasil, maio 2004. SBC.

[11] M. P. Barcellos, H. H. Muhammad, and A. Detsch. Simmcast: a Simulation Tool for Multicast Protocol Evaluation. In *XIX Simpósio Brasileiro de Redes de Computadores, SBRC 2001*, volume 1, pages 418 – 433, Florianópolis, Brasil, May 2001.

[12] P. Veríssimo and L. Rodrigues. *Distributed Systems for System Architects*. Kluwer Academic Publishers, 2001.

[13] R. Munaretti and M. P. Barcellos. Modelagem de Falhas para Simulação de Sistemas Distribuídos. In SBC, editor, *II Escola Regional de Redes de Computadores, ERRC 2004*, volume 1, Canoas, Brasil, julho 2004. SBC.

[14] B. Venners. *Inside the Java 2 Virtual Machine*, chapter 20, Thread Synchronization, page 498;504. McGraw-Hill, 2nd. edition, 1999.