Gustavo Luiz Ferreira Walbon
Antonio Ribeiro Alves Júnior
William Takahashi

Agentes Móveis e Simulação Distribuída

Itajubá - MG

16 de Outubro de 2009

Gustavo Luiz Ferreira Walbon
Antonio Ribeiro Alves Júnior
William Takahashi

Agentes Móveis e Simulação Distribuída

Monografia referente ao trabalho de diploma do curso de Engenharia de Computação da Universidade Federal de Itajubá

Orientador:
Prof. Dr. Edmilson Marmo Moreira

Universidade Federal de Itajubá - Unifei
Instituto de Engenharia de Sistemas e Tecnologias da Informação
Engenharia da Computação

Itajubá - MG

16 de Outubro de 2009

Sumário

Lista de Figuras

1 Introdução

1.1 Simulação

A simulação é uma técnica que permite prever e visualizar o comportamento de sistemas reais a partir de modelos matemáticos. As aplicações da simulação abrangem diversos benefícios, tais como: a possibilidade de antever possíveis problemas ou comportamentos indesejáveis de um sistema, auxílio na tomada de decisão sem a necessidade de intervir no sistema real, facilidade na manipulação e alteração dos modelos, economia de recursos (físicos e financeiros) durante a tomada de decisões, dentre outros.

Para utilizar a simulação é necessário construir e analisar modelos que represente o sistema. Os modelos podem ser classificados de diferentes formas. Uma classificação pode ser considerada verificando a influência ou não de variáveis aleatórias no sistema. Os sistemas são ser representados por um modelo determinístico, quando estes podem ser considerado totalmente livre de aleatoriedade, ou estocásticos, quando estes consideram aleatoriedade.

Os modelos que descrevem o comportamento através do tempo podem ser classificados como contínuos e discretos no tempo. Nos modelos de estados contínuos, as variáveis de estados variam espontaneamente. Já nos modelos de estados discretos, as mudanças ocorrem em pontos específicos e descontínuos do tempo.

Este trabalho enfoca os modelos estocásticos e de estados discretos, uma vez que eles são os que melhor representam modelos de sistemas computacionais.

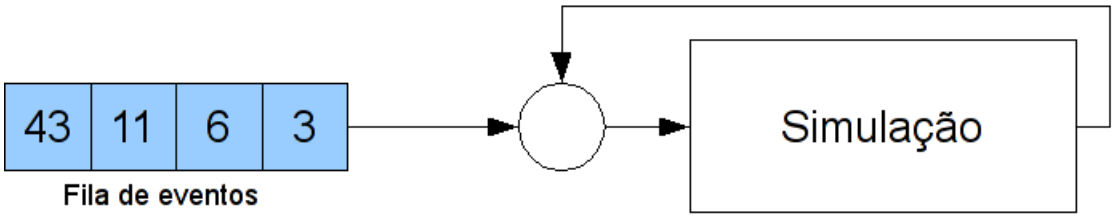

Figura 1: Simulação Sequencial.

Um sistemas de simulação sequencial, onde uma única máquina executa toda a simulação, pode ser retratado como uma fila de eventos aguardando para serem tratados. Cada evento possui o seu tempo de execução, como pode ser visto na figura **??**, que deve ser obedecido para garantir consistência do resultado.

Neste modelo sequencial, o sistema responsável pela simulação retira o próximo evento da fila de execução para tratá-lo. Ao fim do processamento, um próximo evento é retirado da fila, e isto se repete até o final de lista de eventos futuros. O tratamento de um evento pode ou não resultar dados que sejam necessários em um processamento futuro.

1.2 Simulação Distribuída

A simulação é um processo que apresenta um custo computacional muito alto, tendo em vista a grande quantidade de dados que devem ser processados e a complexidade dos modelos matemáticos empregados. Esses fatores em conjunto podem encarecer computacionalmente o sistema, levando à ineficiência da simulação.

Uma das formas encontradas para se solucionar estes problemas foi dividir o tratamento dos diversos eventos entre vários processadores de uma mesma máquina paralela ou sobre um sistema distribuído, dando origem assim à Simulação Distribuída.

Distribuindo os eventos, reduz-se o tempo gasto pelos programas de simulação, mas, em contrapartida, novas situações necessitam de observação devido às carac-

terísticas deste tipo de aplicação. É preciso sanar os problemas com a sincronização dos processos, sobrecarga da rede de comunicação, necessidade de balanceamento de carga do sistema, dentre outros.

Em um sistema de Simulação Distribuída, três estruturas devem ser observadas no desenvolvimento da simulação orientada à eventos:

- As variáveis que descrevem os estados do sistema;
- Uma lista de eventos futuros, que contém os eventos a serem executados;
- Um relógio Global, que controla o progresso da simulação.

Os eventos devem ser executados obedecendo o seu *timestamp*. O programa de simulação deve remover repetidamente da fila o evento com a menor marca de tempo e executá-lo. Assim que um evento é retirado da fila de execução, o relógio global avança para o tempo de ocorrência do evento. Esse mecanismo garante que todos os eventos sejam executados obedecendo a ordem cronológica do tempo de simulação. Porém, em se tratando de um sistema distribuído, não há como haver uma fila única de eventos. Portanto, o sistema passa a ser dividido em n processos denominados $p_1, p_2, \ldots, p_n$, cada um representando um processo do sistema real. Novos mecanismos devem ser incorporados ao sistema de simulação para garantir que cada evento seja executado na sua devida ordem.

Para cada processo lógico é atribuído um relógio que indica o seu progresso na simulação. A comunicação entre os processos se dá através mensagens, uma vez que não há áreas de memória compartilhadas entre os processos. Estas mensagens são também responsáveis pela sincronização do sistema. Caso algum evento e_b venha a ocorrer antes de um segundo evento e_a, e sendo $a < b$, tem-se assim um erro de causa e efeito. Como em um sistema real nunca existirá tal situação, isto caracteriza uma inconsistência na simulação.

Os conceitos de sincronização de processos levaram ao desenvolvimento de protocolos, classificados como conservativos ou otimistas, para garantir a sincro-

nização dos processos da simulação distribuída, evitando ou corrigindo erros de causa e efeito (**??**).

1.3 Objetivos

Este trabalho tem como objetivo avaliar a viabilidade de se implementar um protocolo de sincronização de simulação distribuída utilizando agentes móveis. Para isto é empregada a utilização da biblioteca de agentes móveis *Aglets* para a implementação do protocolo *Rollback* Solidário.

1.4 Organização da Monografia

Os capítulos seguintes abordam uma leve explicação do funcionamento dos protocolos de sincronização de simulação distribuída e de como o trabalho foi abordado para a sua implementação.

O capítulo dois trata dos protocolos de sincronização de eventos em simulação distribuída. Ele inicia com a explicação das principais diferenças entre protocolos conservativos e otimistas, e em seguida traz o princípio básico de funcionamento dos protocolos *Time Warp* e *Rollback* Solidário.

No terceiro capítulo são apresentados os agentes móveis e a biblioteca *Aglets*, assim como o seu funcionamento no contexto computacional.

O quarto capítulo trata a implementação do protocolo *Rollback* Solidário sobre a biblioteca *Aglets*, assim como o seu funcionamento.

Por fim, o capítulo número cinco discute as conclusões obtidas e as possibilidades de continuação deste trabalho.

2 Protocolos de Sincronização de Simulação Distribuída

2.1 Protocolos Existentes

Como foi discutido no capítulo anterior, com a necessidade da sincronização de processos que executam em ambientes diferentes, já que as máquinas tem seus próprios espaços de memória e relógios físicos, faz-se necessária a utilização de ferramentas para sincronizar o sistema. Para isto, os protocolos para a computação distribuída foram criados.

Este capítulo apresenta uma breve discussão dos protocolos conservativos e dos protocolos otimistas, além de apresentar o

2.2 Protocolos Conservativos

Os primeiros autores a tratar os problemas de sincronização nos programas de simulação foram Chandy e Misra(1979) e Bryant(1977), assim batizado esse tipo de algorítmo de CMB em homenagem(**????**). Esse método trata os processos em uma linha de tempo cronológica, não deixando nenhum processo ocorrer fora de seu tempo de execução. Por sua vez, apresenta um problema quanto a determinar se é seguro ou não a execução de um processo, podendo provocar *deadlock* no sistema.

No CMB, os canais de comunicação entre os processos já são previamente especificados antes da simulação, ou seja, um processo só se comunicará com outro, caso um canal já tenha sido programado anteriormente. Neste tipo de protocolo, como os canais de comunicação entre os processos são previamente conhecidos, é possível determinar o tempo lógico dos canais de comunicação entre os processos vizinhos, pois cada mensagem é rotulada de acordo com o *local virtual time* (LVT) de sua máquina.

Misra(1986) sugeriu um mecanismo para resolver o problema dos *deadlocks* provocados por canais que possuem LVTs desatualizados. Seu mecanismo utiliza mensagens nulas para o cálculo do LVT. No entanto, isto pode causar a sobrecarga no sistema e lentidão na rede. Uma variação desta solução seria o envio de mensagens nulas sob demanda, com frequência dada por um temporizador ou quando houver canais com fila vazia.

Há outros protocolos conservativos existentes: SRADS (**??**) , Appointments (**??**) , Turner Carrier-null Scheme (**??**) e SPaDES/Java(**??**) são alguns exemplos.

2.3 Protocolos Otimistas

O que difere basicamente um protocolo otimista de um conservativo, é o modo como ele trata os erros de causa e efeito. Enquanto os protocolos conservativos previnem esses erros, os protocolos otimistas permitem que essa regra seja violada, provendo mecanismos para retornar (*rollback*) a simulação e reconstruí-la de forma consistente.

Assim como nos protocolos conservativos, os processos nos protocolos otimistas se intercomunicam através de mensagens, uma vez que estes não possuem dados compartilhados entre si. Porém, diferente dos protocolos conservativos, nos protocolos otimistas não existe uma garantia que uma mensagem vai ser entregue na mesma ordem em que foi enviada, e também não há como garantir que uma mensagem não seja recebida com um *timestamp* menor que o LVT daquele processo.

Quando uma mensagem chega à um processo com um tempo inferior ao seu LVT, ela é denominada *straggler*. Quando uma mensagem *straggler* surge, a situação é tratada com um *rollback*. Este, por sua vez, deve ser capaz de restaurar a computação, retornando o processamento para um estado global consistente, para poder processar a mensagem e dar continuidade à simulação.

2.4 Time Warp

Um dos mais conhecidos protocolos otimistas é o Time Warp, que possui algumas implementações, tais como *Jade Time Warp* (**??**), o sistema *SPEE-DES* (*Synchronous Parallel Environment for Emulation and Discrete Event Simulation*)(**??**), o *WARPED* (**??**) e o *Georgia Tech Time Warp (GTW)* (**??**). O *Time Warp* foi originalmente proposto por Jefferson(1985).

O *Time Warp* pode ser dividido em duas partes distintas: o controle local, interno a cada processo, o qual é responsável por garantir que a execução dos eventos seja feita em ordem cronológica dentro de cada processo; e o controle global, destinado ao gerenciamento de memória e ao cálculo do *Global Virtual Time*(GVT)(JEFERSON, 1985).

O comportamento da simulação no protocolo *Time Warp* é semelhante ao de um programa sequêncial. Os eventos internos à um processo são organizados em uma lista, respeitando os seus *timestamps*. Estes eventos são retirados da lista para serem tratados.

A comunicação entre os processos é feita através de troca de mensagens. As mensagens, assim como os eventos, também são rotuladas com uma marca de tempo.

Os eventos processados podem gerar mensagens a serem enviadas a outros processos. Estas mensagens, por sua vez, geram eventos que devem ser alocados na lista de eventos futuros para serem tratados, respeitando o seu *timestamp*.

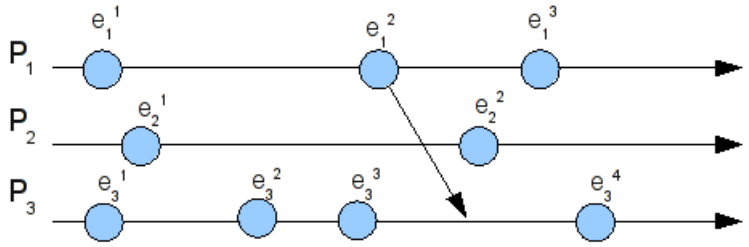

Figura 2: Mensagem strangler.

Uma vez que um processo recebe uma mensagem com um *timestamp* inferior ao valor do seu LVT, tem-se uma situação de inconsistência no sistema. Como um protocolo otimista, o *Time Warp* deve ser capaz de retornar a simulação a fim de reconstruí-la, agora considerando também a mensagem que havia chegado fora da ordem.

No exemplo da figura **??**, o evento e_1^2 envia uma mensagem com LVT igual a 2 para um processo que já está no tempo 3. Isso gera uma mensagem *straggler*, pois o evento e_3^3 só deveria ser executado após o recebimento da mensagem. Para a correção, o protocolo deve garantir que o evento e_3^3 seja desfeito e que ele somente seja executado depois do tratamento da mensagem enviada pelo evento e_1^2.

2.4.1 Rollback primário e Rollback secundário

Assim que uma mensagem *straggler* é identificada por um processo, este retorna a um tempo lógico imediatamente inferior ao *timestamp* da mensagem recebida. Isto é feito para que os dados provenientes da mensagem sejam tratados, mantendo a ordem natural de execução da simulação.

Porém, um processo que necessitou fazer um *rollback* pode ter enviado mensagens a outros processos em um tempo superior ao seu tempo lógico de retorno. Todas essas mensagens enviadas devem ser desconsideradas pelo sistema, e os processos que porventura tenham tratado eventos gerados por estas mensagens devem também retornar, podendo gerar novos *rollbacks* no sistema.

Um *rollback* que foi gerado devido ao recebimento de uma mensagem *straggler* é chamado de *rollback* primário, já um retorno causado por efeito de um outro *rollback* é chamado de *rollback* secundário.

2.4.2 Anti-Mensagem

Ao realizar um *rollback*, um processo deve verificar se foram enviadas mensagens com tempo lógico superior ao valor do seu tempo de retorno. Para toda mensagem enviada, o processo deve mandar uma anti-mensagem, indicando que as mensagens anteriores devem ser desconsideradas.

Ao receber uma anti-mensagem, um processo pode encontrar três situações distintas:

- O evento gerado pela mensagem a ser cancelada já foi tratado. Então deve-se realizar um retorno (*rollback* secundário) para um tempo lógico inferior ao recebimento da mensagem, e fazer, se necessário, também o envio de anti-mensagens;
- O evento gerado ainda está na fila a espera da execução. Neste caso o processo apenas exclui este evento da lista de eventos futuros;
- Anti-mensagem chegar ao processo antes da mensagem. Caso essa situação aconteça, o processo armazena a anti-mensagem para, quando a mensagem chegar, eliminar as duas.

2.5 RollBack Solidário

Assim como o *Time Warp*, o *Rollback* Solidário é um protocolo otimista, entretanto, ele apresenta diferenças significativas na maneira com que os retornos são realizados para sincronizar os processos durante a simulação distribuída.

Uma primeira diferença que pode ser citada é que no protocolo *Rollback* Solidário, quando um erro de causa e efeito é identificado durante a simulação, todos os processos executam em conjunto um *rollback* para um *checkpoint* global consistente, enquanto no protocolo *Time Warp*, em um primeiro instante, apenas o processo que identificou o erro de causa e efeito realiza o *rollback*.

Checkpoints são estados de um processo armazenados em um mecanismo "persistente"de memória para a recuperação de um sistema em caso de falhas(**??**). Um checkpoint local é o conjunto de dados de um processo num determinado tempo da simulação e um *checkpoint* global é um conjunto de *checkpoints* locais, sendo um para cada processo da simulação.

Para que um *checkpint* global possa ser utilizado como um ponto de retorno no caso de um erro na simulação, deve-se garantir que esse checkpoint global seja consistente. Para ser considerado consistente, um *checkpoint* global deve estar associado a um corte global consistente e nenhum *checkpoint* do conjunto deve possuir relação de dependência com outro *checkpoint* do mesmo conjunto.

Estas diferença na maneira de se realizar o retorno dos procesos dispensa a necessidade de se enviar anti-mensagens, que poderiam provocar *rollbacks* em outros processos, evitando assim a ocorrência de *rollback* em cascata.

2.6 O comportamento geral do protocolo Rollback Solidário

No decorrer de uma simulação, utilizando o protocolo *Rollback* Solidário, assim como no *Time warp*, os processos se intercomunicam através de mensagens. Porém, ao identificar um erro de causa e efeito, ou seja, ao se receber uma mensagem *straggler*, o sistema se comporta da seguinte forma:

- O processo que recebeu a mensagem *straggler* identifica o estado salvo (*checkpoint*) com tempo lógico imediatamente anterior ao *timestamp* da mensagem.

- O sistema se encarrega de selecionar o *checkpoint* global consistente que contenha este estado identificado e que cause a menor sobrecarga durante o *rollback* (mínimo possível de eventos para se retornar).

- Toda a simulação realiza o retorno à este *checkpoint* global consistente de maneira uniforme.

2.7 Recuperação dos Estados Durante um Rollback

A eficiência do protocolo *Rollback* Solidário está diretamente ligada em como ele retorna à um *checkpoint* global consistente após identificar uma inconsistência na simulação. O *checkpoint* global consistente a ser selecionado deve gerar o menor custo para o sistema. Para isto, deve-se garantir que os processos voltem a menor distância possível necessária para recompôr o processamento.

Considerando o exemplo da figura **??**, sabe-se que em relação a seus tempos lógicos tem-se $t'' < b' < t' < t$. Assim sendo, supondo que o processo p_1 receba uma mensagem no tempo t, gerando um evento que deveria ter sido executada no tempo t', ocorre que neste momento o sistema deve retornar ao último *checkpoint* global consistente que contenha o processo b', que neste caso é o corte B. Porém, supondo que o processo P_1 recebe, no tempo t, uma mensagem que gera um evento a ser processado no tempo t'', anterior ao evento b' , não pode-se admitir o corte B como linha de recuperação, devido ao tempo lógico de b' ser maior que o tempo lógico de t''. Acontece aí então um *rollback* para a linha de recuperação A.

Nota-se, nessa situação, que nem todos os processos precisam voltar para a mesma linha de recuperação. No exemplo da figura **??**, tanto os processos P2 e P3 podem permanecer com os mesmos *checkpoints* do corte B, uma vez que não houve troca de mensagens por parte dos processos entre os cortes A e B.

Para garantir a eficiência do protocolo, deve-se garantir que a linha de recuperação preserve ao máximo os eventos já processados. Para isso, é necessário

armazenar várias linhas de recuperação que passem por um mesmo *checkpoint*.

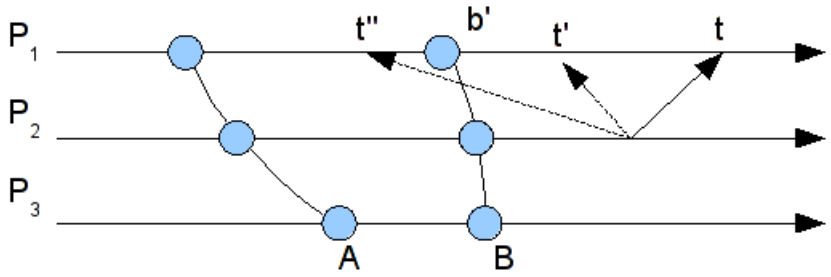

Figura 3: Linhas de recuperação.

2.8 Uma abordagem utilizando checkpoints síncronos

Mecanismos de *checkpoints* síncronos são mecanismos que exigem que os processos realizem seus checkpoints em conjunto para a obtenção de um *checkpoint* global consistentes. Para o protocolo *Rollback* Solidário utilizando *checkpoints* síncronos são apresentadas as extenções dos os algorítimos *Sync-and-Stop*(**??**) e *Chandy-Lamport*(**??**).

2.9 Rollback Solidário utilizando mecanismos de checkpoints Semi-Síncronos

Para a construção da linha de recuperação na abordagem semi-síncrona, um processo, denominado processo observador, fica encarregado de receber os vetores de dependência associados aos respectivos *checkpoints* de cada processo e identificar o *checkpoint* global consistente para onde a simulação deve retornar.

Ao realizar um *checkpoint*, um processo envia ao observador um vetor com as suas dependências. Com isto, o processo observador constrói um matriz quadrada M de ordem n, onde n é o número de processos da simulação. Cada linha i da

matriz é o vetor de dependência de um processo p_i, que contém as informações de dependência do último *checkpoint* realizado por esse processo p_i.

A matriz é iniciada com o primeiro *checkpoint* de cada processo. Como todos os processos no início da simulação estão num mesmo evento inicial, a matriz M é iniciada como uma matriz identidade.

A diagonal principal da matriz M poderá indicar uma linha de recuperação da simulação. Para que esta linha de recuperação seja válida, todos os elementos de uma coluna que não fizerem parte da diagonal principal da matriz devem possuir valores menores do que o elemento pertencente à diagonal.

Para garantir que todas as possíveis linhas de recuperação sejam identificados um novo vetor de dependência é recebido pelo processo observador, o último vetor referente ao processo que realizou o *checkpoint* não deve ser descartado, mas sim armazenado para futuras consultas.

A solução para se armazenar os diversos vetores está em utilizar, ao invés de matriz quadrada uma lista encadeada para cada processo, cada uma contendo os vetores de dependência.

Ao receber um aviso de *rollback*, proveniente de algum processo, cabe ao processo observador percorrer as listas e identificar os *checkpoints* adequados para constituir a linha de recuperação correspondente.

3 Agente Móveis

Um agente é uma entidade autônoma e independente que executa sobre um determinado contexto computacional. O termo agente descreve uma abstração ou um conceito de modelo de *software*. Em outras palavras, agentes são partes executáveis de programas que durante o seu ciclo de vida, além de executar as suas próprias tarefas, pussuem a capacidade de interagirem com outros agentes e têm seu ciclo de vida limitado a um ambiente de execução, denominado contexto.

Conforme ilustrado na figura **??**, uma vez um agente estando em execução em um determinado contexto, ele executa suas tarefas específicas. Mas, além disso, os agentes têm a possibilidade de interação com os demais agentes em execução (ilustrada pelas linhas tracejadas).

O contexto aqui chamado é uma plataforma onde o agende executa. Um agente por si só não possui a propriedade de execução, para isto ele necessita de um ambiente propício, onde seu ciclo de vida possa ser controlado (criação e exclusão do agente) e este execute suas tarefas. Uma outra maneira de se enxergar o contexto de execução do agente seria a de um *framework* onde os agentes poderiam ser instalados e executados. Sendo assim, um agente existe apenas naquele determinado contexto no qual ele foi criado, seu ciclo de vida (criação, execução e exclusão) se restringe sempre à plataforma de execução.

Um tipo em especial de agente são os Agentes Móveis. Para essa tipo de agente, uma característica importante a destacar é que ele pode-se mover para um contexto diferente do qual foi originalmente criado. Isso significa que o agente

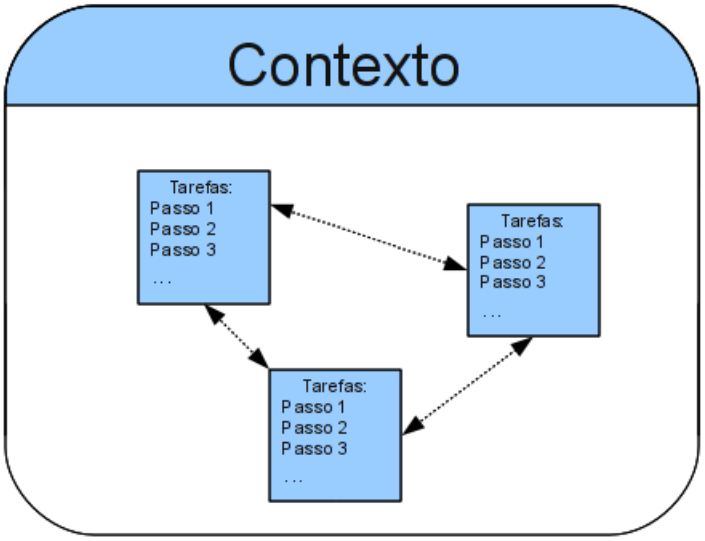

Figura 4: Agentes em um determinado contexto.

deixou de ter como limitação aquela plataforma à qual ele existia num momento inicial e ganhou a possibilidade de se mover para outros contextos, caso isso seja necessário durante o seu ciclo de vida.

Algumas utilidades que ajudam a ilustrar os benefícios de se migrar um agente para um contexto diferente do original podem ser citadas a seguir:

- Aproximação do processo com a sua base de dados, diminuindo o tráfego de dados pela rede;
- Mudança da máquina a ser utilizada pelo processo, seja por motivos de manutenção ou para se aproveitar de um *hardware* mais poderoso ou mesmo um periférico específico;
- Distribuição do processo entre vários *hosts*, etc.

Conforme ilustrado na Figura **??**, um agente que existia previamente em um contexto A pode, a qualquer momento de sua vida, migrar para um outro contexto B. Para isto basta que exista um meio de ligação entre os dois *hosts* (uma rede

de computadores) e a existência do contexto anfitrião propício a receber o agente móvel.

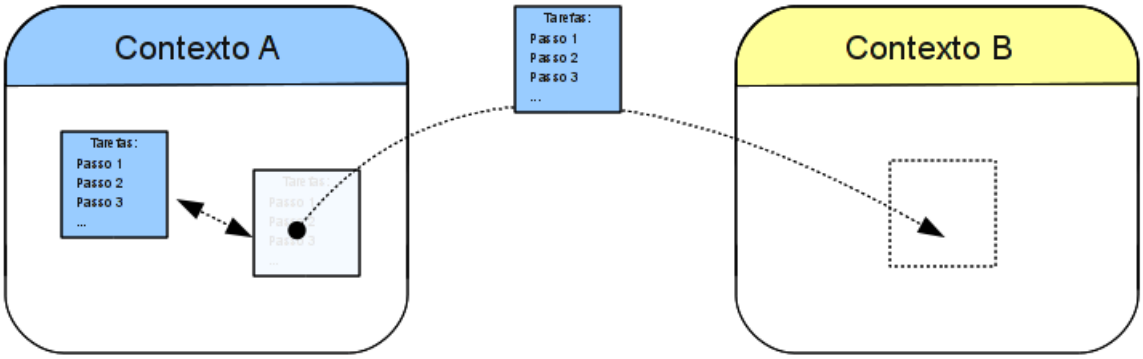

Figura 5: Agente migrando para um contexto diferente do original.

3.1 Mobilidade Forte e Mobilidade Fraca

Durante uma migração um agente móvel leva consigo todos os dados referentes à execução, além do seu código executável. Em sua concepção original, um agente móvel, ao migrar, deve interromper sua execução no ponto onde ela estiver, encapsular todos os dados referentes ao processo, migrar para o seu destino e, ao chegar, continuar o processamento do ponto onde ele parou. A este modelo de migração, onde o processamento do agente continua no exato ponto onde parou antes da migração, dá-se o nome de Mobilidade Forte.

Um segundo tipo de migração é a Mobilidade Fraca. Na mobilidade fraca, ao se migrar para um novo contexto, o agente encapsula todas as variáveis globais do processo, porém não continua o processo do exato ponto onde parou, mas sim recomeça o processamento desde o início.

Para melhor ilustrar esse comportamento. Considere, no seguinte exemplo, que para o dado código, exatamente após a terceira interação, o agente mova do contexto A para o contexto B:

Sendo assim, existirá dois comportamentos distintos para migrações baseadas

```
for(int i = 0 ; i < 5 ; i++) {
        printf("%i ," ,i);
}
```

Tipo	Contexto A	Contexto B
Mobilidade Fraca	0, 1, 2,	0, 1, 2, 3, 4,
Mobilidade Forte	0, 1, 2,	3, 4,

Tabela 1: Saída de terminal para o código citado em diferentes tipos de migração.

em Mobilidade Forte e mobilidade Fraca, conforme a tabela **??**.

Como pode ser notado com base nas saídas de terminal para ambos os tipos de mobilidade, na Mobilidade Fraca a iteração é interrompido antes da migração, mas recomeça a partir do zero ao chegar no seu novo contexto. Já no caso da Mobilidade Forte a iteração, após ser interrompida retoma o seu processamento no novo contexto a partir do exato ponto onde este foi interrompido no contexto original. A biblioteca de agentes móveis *Aglets*, utilizada para esse trabalho, suporta apenas Mobilidade Fraca. Mais detalhes sobre esse assunto serão discutidos posteriormente, na seção sobre *Aglets*.

3.2 Agentes Móveis em Java e a Biblioteca Aglets

Por possuir um código que é executado sobre uma máquina virtual, o Java naturalmente provêem uma camada de abstração para o código a ser executado. Isso significa que, tendo o interpretador Java devidamente instalando em computadores com diferentes sistemas operacionais ou até mesmo baseados em diferentes arquiteturas, têm-se a garantia de que o código será executado da mesma forma, independente do hardware ou ou da plataforma em cada um dos *hosts*. Essa característica facilita a criação de agentes móveis, uma vez que, tendo um processo em execução, caso seja necessário transferi-lo para um segundo contexto, não existe a preocupação de traduzir o código. Para isto, basta simplesmente encapsular as

variáveis presentes na máquina virtual Java, além do código Java executável, e transferí-los para a máquina destino.

Para abstrair todo o processo de encapsulamento dos dados de um agente e demais processos envolvidos no controle de seu ciclo de vida e mobilidade, é empregado a utilização de bibliotecas que já provêem todas essas funcionalidades para se trabalhar com agentes móveis em Java. Uma dessas bibliotecas, a utilizada neste trabalho, é a *Aglets* (*Agent Applet*). *Aglets* é uma biblioteca desenvolvida pela IBM Tokio *Research Laboratory*, que possibilita a criação de agentes móveis em Java. Um *aglet* é um agente Java capaz de executar suas tarefas e migrar de um *host* para outro, além de prover ferramentas para multiplicação (clonagem), comunicação entre *aglets* e gerenciamento do ciclo de vida (criação e exclusão de um *aglet*).

Um *aglet* possui o seu ciclo de vida restrito à plataforma de *aglets* provida para o seu funcionamento. Como padrão, a biblioteca *Aglets* utiliza o servidor *Tahiti* como plataforma de execução dos agentes *aglets* (Figura **??**)(**??**).

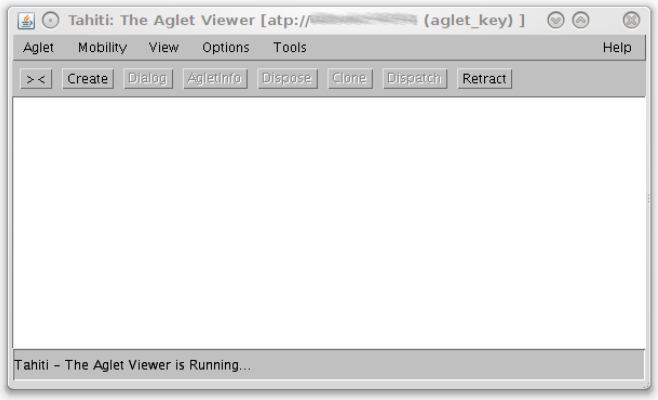

Figura 6: Servidor de *Aglets Tahiti*.

O *Tahiti* provê a interface para se trabalhar com *aglets*. Através do servidor *Tahiti* pode-se criar e manipular os agentes. Existe também a possibilidade de se utilizar o *Tahiti* sem a interface gráfica através do console(Figura **??**).

Todo agente *aglet* é um objeto derivado da classe pai abstrata *Aglet*. Ao se

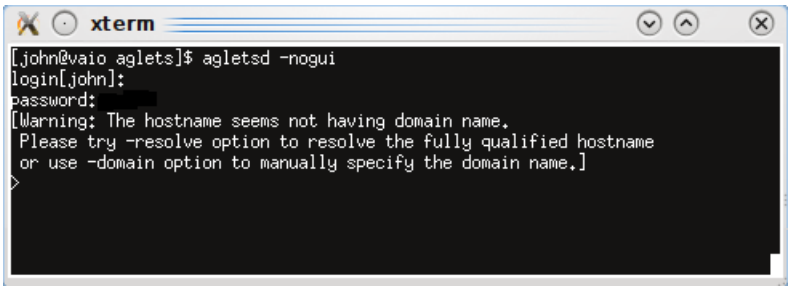

Figura 7: Servidor *Tahiti* por linha de comandos.

criar um *aglet*, os métodos pré-definidos são sobrescritos com o código conveniente, a fim de se criar a aplicação desejada. Uma vez desenvolvidos os agentes *aglets*, utiliza-se o servidor *Tahiti* para iniciar o ciclo de vida dos agentes.

3.3 Ciclo de Vida Aglets

Todo agente possui um ciclo de vida definido dentro de seu contexto. Conforme ilustrado na Figura **??**, um *aglet* possui cinco principais fases durante o seu ciclo de vida:

- Criação: Ocorre uma única vez em todo o ciclo de vida do *aglet*. Ao ser criado, um *aglet* invoca o método *onCreation*() e executa o conteúdo deste método. O método *onCreation*() é um método abstrato da classe pai *Aglet* e deve ser sobrescrito pelo agente.

- Execução: A execução (método *run*()) de um *aglet* acontece em três momentos distintos em seu ciclo de vida: após sua criação (logo após a execução do método *onCreation*()), após uma migração (assim que o agente chega ao *host*) e após uma clonagem, executado apenas pelo novo agente (o original continua o seu ciclo de execução natural). A execução é invocada através do método *run*(), também herdado da classe pai *Aglet* e que também deve ser sobrescrito.

- Clonagem: Após uma clonagem, o agente original cria uma cópia idêntica de si. Esta cópia executa os métodos referentes ao clone *listener* do agente e em seguida executa o método *run*() referente. Após criado, o clone segue um ciclo de vida completamente independente do agente original.

- Migração: Um método que se move de um contexto para outro carrega consigo todas as informação do momento anterior à migração. Ao chegar em seu novo *host*, o agente que executou a migração trata os eventos referentes ao *Mobility Listener* e em seguida executa o método *run*(). Como o método *run*() é executado a partir de seu início, toda a execução que porventura estava sendo executada dentro deste método antes da migração será refeita. Por esse motivo é que a migração de um agente *Aglet* é considerada do tipo Mobilidade Fraca.

- Exclusão: O final da vida de um *aglet* não se dá ao final da execução do método *run*(), mas sim deve ser explicitamente chamado. Caso contrário, o agente permanece no seu referente contexto.

3.4 Comunicação Entre Aglets

Agentes comunicam entre si através de mensagens. Essas mensagens são enviadas de um agente para outro através de *proxys*. Um *proxy* pode ser visto neste caso como uma interface de comunicação entre agentes, que proporciona transparência e segurança na comunicação entre os *aglets*. Para que um *aglet* se comunique com outro, basta ter o seu *proxy* remoto, mascarando assim a real localização do agente.

Todo agente possui um identificador único e é através desse identificador e da URL onde o agente se encontra que se obtém o *proxy* possibilitando a comunicação.

O método `handleMessage(Message msg)` é responsável por tratar as mensagens recebidas pelos agentes. Esse método recebe como parâmetro um objeto do tipo *Message* que contém os dados da mensagem. O objeto Message possui dois

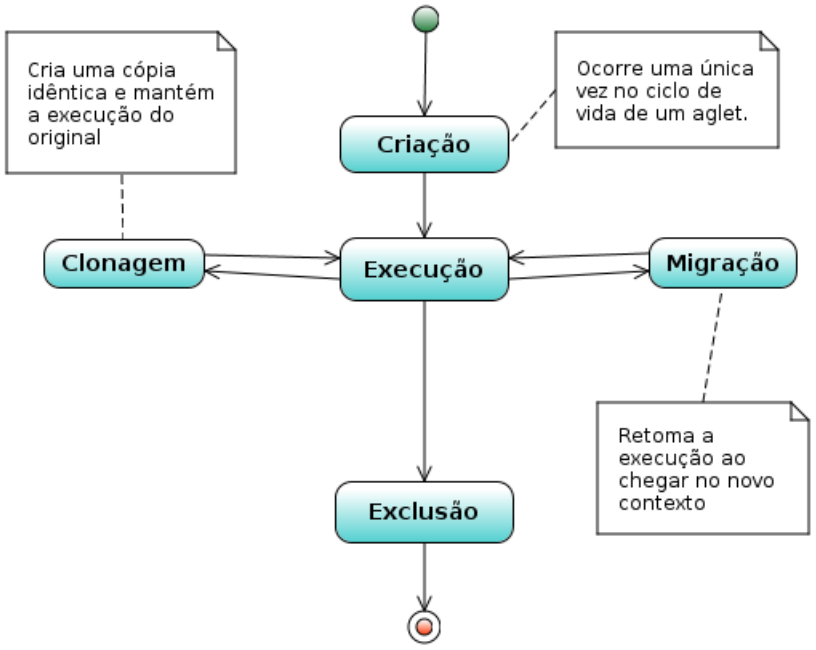

Figura 8: Ciclo de vida de um *aglet*.

principais atributos como ilustra o diagrama da Figura **??**. O atributo *kind* é uma *string* e contém o tipo da mensagem, um rótulo. O atributo *arg* é um objeto que pode ser enviado através da mensagem. Este objeto deve ser serializável para que possa ser enviado através de uma mensagem.

Ao receber uma mensagem, o método *handleMessage*(...) trata a mensagem com base em seu rótulo (*kind*). Toda a comunicação entre os agentes, mesmo os que estão no mesmo contexto, é feita através de mensagens e, consecutivamente, através dos *proxys* dos *aglets*. Veja a Figura **??**

3.5 Considerações Finais

As propriedades de mobilidade e independência dos agentes móveis proporcionam características que podem ser exploradas na implementação de aplicações distribuídas. O fato de um agente poder se mover por diferentes *hosts* em tempo

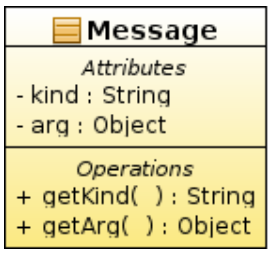

Figura 9: Classe Message.

de execução, por exemplo, pode ser usado como forma de balanceamento de cargas, ou simplesmente para manter processos que trocam muitas mensagens fisicamente perto, em uma mesma rede, agilizando assim o tráfego de dados na rede. A facilidade proporcionada para a troca de mensagens e envio de objetos encapsulados dentro dessas mensagens também pode ser utilizada para enviar eventos de um nó de uma rede de processamento para outro, ou enviar eventos que foram gerados em um processo para serem tratados por outro processo.

É com base nessas e em outras características de agentes móveis e da biblioteca *Aglets* que a implementação do protocolo *Rollback* Solidário deste trabalho foi desenvolvida, a fim de se avaliar a viabilidade da implementação de um software de simulação distribuída utilizando das propriedades dos agentes móveis e, mais especificamente neste caso, da linguagem Java e da biblioteca *Aglets*.

4 Estrutura e Implementação

Existem já implementadas versões do protocolo *Rollback* Solidário utilizando-se de outras bibliotecas de comunicação e em outras linguagens. A impementação utilizando-se uma biblioteca de agentes móveis implica em algumas características especiais, como a necessidade de haver um servidor de *aglets* em funcionamento em cada estação de trabalho, tal como a intalação da biblioteca de agentes móveis e da máquina virtual Java.

Como já explicado no capítulo sobre agentes móveis, um agente necessita de um ambiente dedicado para execução. Esse ambiente, neste caso específico para a biblioteca *Aglets* o servidor *Tahiti*, deve estar devidamente instalado e em funcionamento em cada um dos hosts que estarão aptos à fazerem parte do sistema. Cada sessão do ambiente Tahiti está vinculado à um URL e à uma porta (por padrão, o *Tahiti* utiliza a porta 4434). Isso permite a utilização de mais de uma contexto em execução em um mesmo host. A possibilidade de se manter mais de um contexto em execução em uma mesma máquina, para uma aplicação de simulação distribuída, a princípio pode não ser uma vantagem. Mas em tempo de implementação e testes iniciais esta característica foi muito explorada. A implementação deste trabalho foi, por diversos momentos, executados sobre um mesmo ambiente computacional com diversos ambientes Tahiti em execução, um em cada porta, a fim de se simplificar os testes iniciais. Uma vez em estágio mais avançado, os mesmo testes foram aplicados dividindo-se os processos em hosts em pontos distintos de uma rede.

Existem outros pontos que podem ser citados como característica a serem exploradas para o caso da possibilidade de co-existência de diferentes contextos *aglets* em uma mesma máquina, como a possibilidade de manter um processo de execução e o observador em um mesmo ambiente computacional, ou mesmo migrar um processo temporariamente para uma máquina, em caso de manutenção ou interrompimento temporário do funcionamento de um dos *hosts*.

4.1 Implementação

Como todo agente aglet deve ser criado a partir de um ambiente Tahiti, a implementação se inicia por um agente superior, encarregado de criar todos os demais agentes do sistema. Este agente, denominado Servidor, fica encarregado de carregar a lista de hosts do sistema, criar os processos e despachá-los para o seu ambiente de execução. Conforme ilustra o diagrama da Figura **??** , a implementação conta com seis principais classes:

- **Servidor**: Responsável por iniciar todo o processo. É esta classe que carrega a lista de *hosts* e despacha os processos aos seus hosts de destino;
- **ProcessoPai**: Classe abstrata, que carrega consigo os dados em comum entre um Processo e o Observador;
- **Processo**: Provém toda a estrutura para se tratar um evento. Um processo possui uma lista de eventos a serem tratados. Todo processo tem a capacidade de se comunicar com outro processo qualquer;
- **Evento**: Um evento é uma classe que provém todo o tratamento para execução do código de simulação distribuída;
- **DadosProcesso**: DadosProcesso é a classe que contém os dados referentes a cada um dos processos em execução, incluso o processo *Observador*;

- **Observador**: A classe Observador implementa todo o tratamento de recebimento dos vetores de dependência, eleição de linhas de recuperação, etc.

Todo o processo de iniciar os agentes, carregar os dados referentes aos *hosts*, carregar a lista de eventos de cada processo e a inicialização do processo observador é feita no mesmo ambiente onde se executa o processo **Servidor**. Os processos de execução e o processo observador só iniciam seu funcionamento após chegarem ao seu ambiente de execução (após migração). Isso garante que a criação de um número elevado de agentes não deixe o sistema, em um momento inicial, sobrecarrega, pois o tratamento dos eventos da lista de eventos futuros estará bloqueado enquanto os processos ainda estiverem no mesmo ambiente o qual foram criados (antes de uma migração).

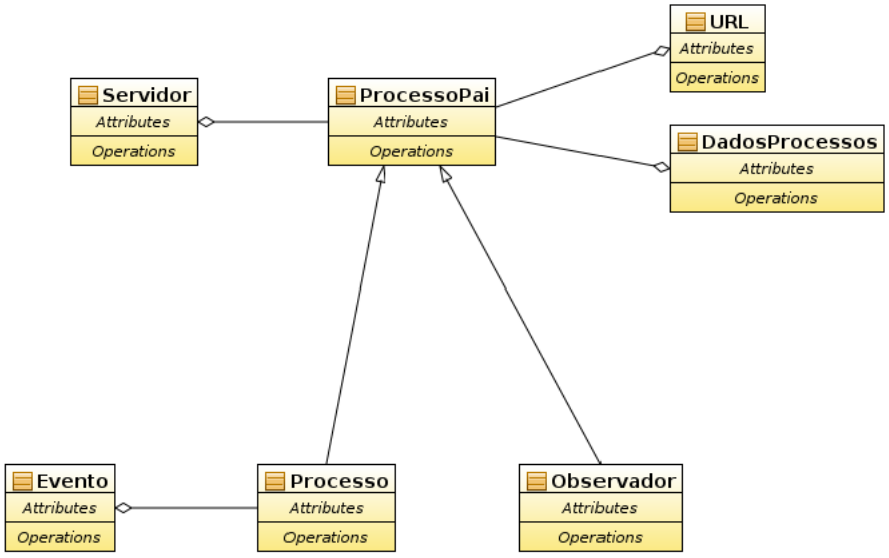

Figura 10: Diagrama de Classes.

Cada um dos *hosts* deve estar aptos a receberem um agente *aglet*, com o servidor *Tahiti* devidamente em execução.

4.1.1 A Classe Servidor

A classe Servidor é a responsável por instanciar todos os agentes do sistema. Essa classe deve ser criada já a partir de um servidor *Tahiti*. Ao iniciar, a classe servidor se comporta conforme ilustrado no diagrama da Figura **??**. A implementação da classe servidor e seus métodos, conforme ilustrado no diagrama da Figura **??**, são descritos a seguir.

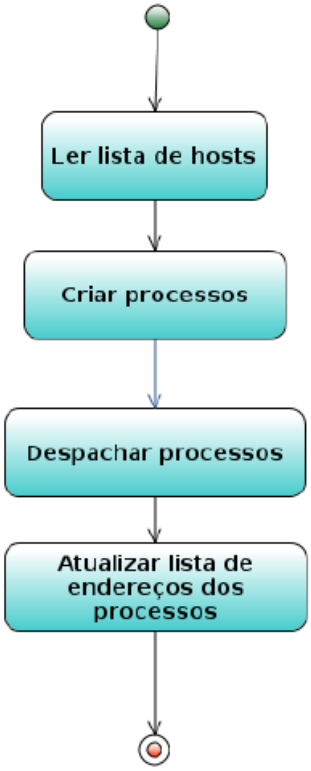

Figura 11: Diagrama de atividades da classe Servidor.

O método ***onCreationb(Object ini)***, derivado da classe pai *Aglet*, é executado assim que o agente servidor é instanciado sobre o servidor *Tahiti*. A execução desse método consiste em carregar toda a lista de *Urls* onde serão despachados os processos para serem executados. Ao fim da leitura da lista de hosts, este método se encerra. Como a execução desse método é única em todo o ciclo de vida do agente Servidor, a lista de endereços dos *Hosts*, assim como a quantidade de

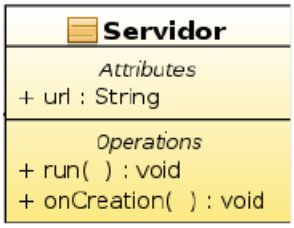

Figura 12: Diagrama da classe Servidor.

processos, não mais será alterada durante a simulação. Em caso de migração de algum processo para um ambiente diferente do ambiente original de simulação, a atualização do novo endereço do processo se dá na lista *DadosProcesso*.

Após a execução do método ***onCreation(...)***, o método ***run()*** assume o controle do agente Servidor. É dentro desse método que os processos são criados (com base na lista de *Urls*. Ao primeiro endereço da lista é atribuído o Processo Observador, os demais endereços são processos de tratamento de eventos. Em um primeiro momento, todos os agentes Processo são criados, e seus dados de controle (*id*, *AgletID* e *LocalProxy*) são armazenados na lista DadosProcessos, para eventuais comunicações. Somente após a criação de todos os processos é que estes agentes são despachados para seus *hosts* de execução. Isso se dá porque cada processo deve conter os dados referentes a todos os processos, inclusive o Observador, para troca de mensagens. Assim que cada agente Processo é enviado para o seu ambiente de execução, os dados relativos aos demais processos são também enviados e, ao chegarem, recompóem a lista *DadosProcesso*.

Ao final do processo de migração de todos os agentes para seus devidos contextos, o agente Servidor encerra sua participação na simulação, e todo o controle a partir desse momento se dá entre os Processos e o Observador.

4.1.2 A classe Processo

A classe Processo é responsável por toda a manipulação dos eventos a serem tratados, tal como a inserção de um novo evento, detecção de mensagem *straggle*,

migração, entre outros, conforme pode ser verificado no diagrama da Figura **??**.

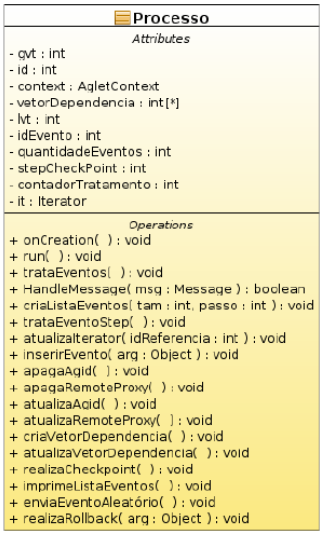

Figura 13: Diagrama da classe Processo.

Como toda classe derivada da classe *Aglet*, a classe Processo implementa os métodos ***run()*** e ***onCreation(...)***. O método ***onCreation(...)*** é responsável por criar a lista de eventos futuros e instalar os escutadores (*listeners*) responsáveis pelos eventos de migração e clonagem. Já o método ***run()*** fica responsável por atualizar o contexto do agente e instanciar o *iterador (iterator)* da lista de eventos futuros. O tratamento dos eventos não é feito dentro do método ***run()*** devido ao estilo de migração dos agentes *aglets*. Como a migração de um *aglet* é baseado em mobilidade fraca, caso o tratamento se desse dentro do método ***run()*** todo o processamento deveria ser refeito caso houvesso uma migração durante a simulação.

A execução do agente processo pode ser dividido em duas partes principais: o tratamento dos eventos da lista de eventos futuro, ilustrado na Figura **??**, e o tratamento do recebimento de mensagens (abordade em uma subcessão específica).

O tratamento dos eventos no loop principal do agente ***Processo*** consiste em retirar o próximo evento da lista de deventos futuros, tratar os dados referentes a esse evento, verificar a criaçào de um novo evento relativo ao processamento atual e retornal ao início do ciclo. Ao finalizar um ciclo de tratamento de eventos, o

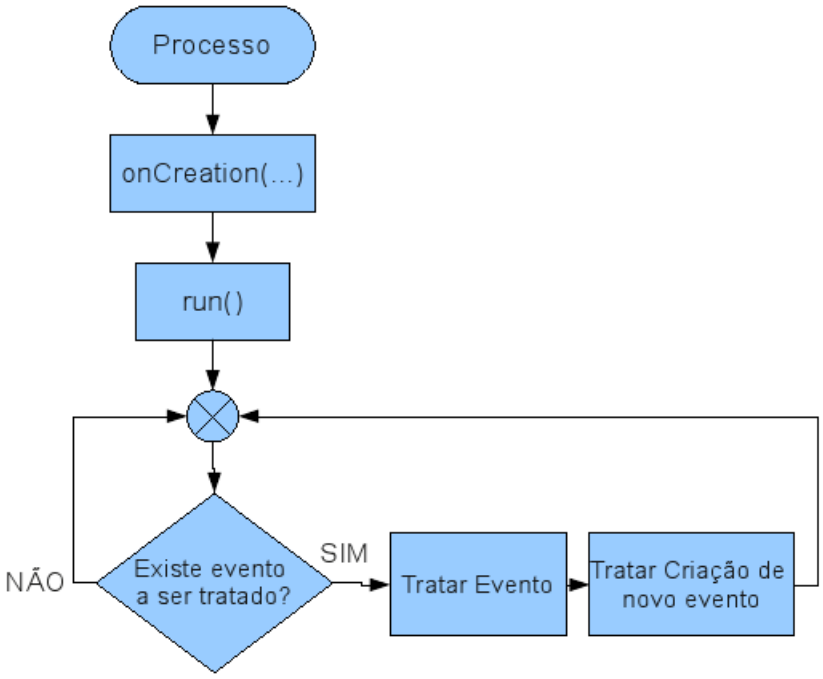

Figura 14: Tratamendo de eventos na classe Processo.

processo envia uma mensagem para si mesmo, indicando o fim do tratamento de um ciclo. Essa mensagem é colocada na fila de mensagens e é processada pelo método ***handleMessagem(...)***, o qual inicia um novo ciclo de tratamento de evento. Esse processo permite a eliminação de um laço do tipo ***while(...)*** ou um laço ***for(...)*** para a execução do ciclo de tratamento de eventos. Caso o tratamento se desse dentro de uma das estruturas de repetição citadas, ocorreria um sério problema em relação ao tratamento das mensagens chegas em tempo de execução (essa abordagem será melhor detalhada na subsessão 5.1.4 , sobre tratamento de mensagens).

Existem dois pontos importantes relacionados ao tratamento de eventos: como é feito o tratamento em relação à mobilidade fraca e como é feita a atualização da lista de eventos futuros em caso de recebimento de um novo evento.

Todo o tratamento da lista de eventos futuros é feita com base em uma variável do tipo Iterator. Essa variável é uma interface que permite as operações básicas de

percorrer uma coleção. Na implementação do agente Processo, a variável Iterator sempre aponta para o próximo evento a ser tratado.

O método atualizaIterator(...) recebe como parâmetro o LVT do processo atual, e atualiza a variável *Iterator* com base nesse valor. Existem dois principais motivos para a atualização desta variável com base no LVT do processo, os quais garantem o tratamento ininterrupto sobre um agente com mobilidade fraca e a atualização da lista de eventos futuros:

- A variável ***Iterator*** é do tipo não serializável, ou seja, o seu valor não suporta uma migração. Para contornar essa característica, o valor da variável é apagado (recebe *null*) antes da migração e é atualizado com base no LVT ao chegar no seu novo contexto de execução;
- Em caso de inserção de um novo evento na lista de eventos futuros, caso esse novo evento seja o vizinho sucessor do atual evento em execução, a variável Iterator apontará para um evento incorreto. Para sanar esse problema, o iterator deve ser atualizado quando essa ocasião acontecer.

Existe uma terceira ocasião onde a atualização do ***Iterator*** é muito importante na simulação: na ocorrência de um *rollback*. Ao ser notificada a realização de um *rollback*, o processo recebe o valor de tempo para o qual deve retornar. Esse valor deve ser utilizado para atualizar o *iterador* da lista de eventos futuros e continuar a simulação.

Existe uma outra gama de métodos implementados que não fazem parte da estrutura do protocolo *Rollback* Solidário, mas que foram inseridas na implementação para dar base de sustentação para testes do funcionamento da implementação. Os métodos ***criaListaEventos(...)*** e ***enviaEventoAleatório(...)*** são, respectivamente, responsáveis por criar uma lista de eventos de um tamanho determinado e enviar um evento aleatório para um processo também aleatório. Esses métodos não fazem sentido para o protocolo em si, mas permitem simular o funcionamento do protocolo em uma ocasião real de simulação.

4.1.3 A classe Observador

A classe Observador implementa os métodos necessários para o recebimento e tratamento dos checkpoints e para a seleção das linhas de recuperação em caso de rollback. A classe Observador, assim como a classe Processo é derivada da classe ProcessoPai que por sua vez é derivada da classe Aglet. Com isso essa classe carrega as características de um agente aglet e implementa o método run(). São implementados também os demais métodos referentes à inserção de novos vetores de dependência, seleção de linha de recuperação e tratamento de rollback. Os métodos implementados são ilustrados no diagrama da Figura (**??**).

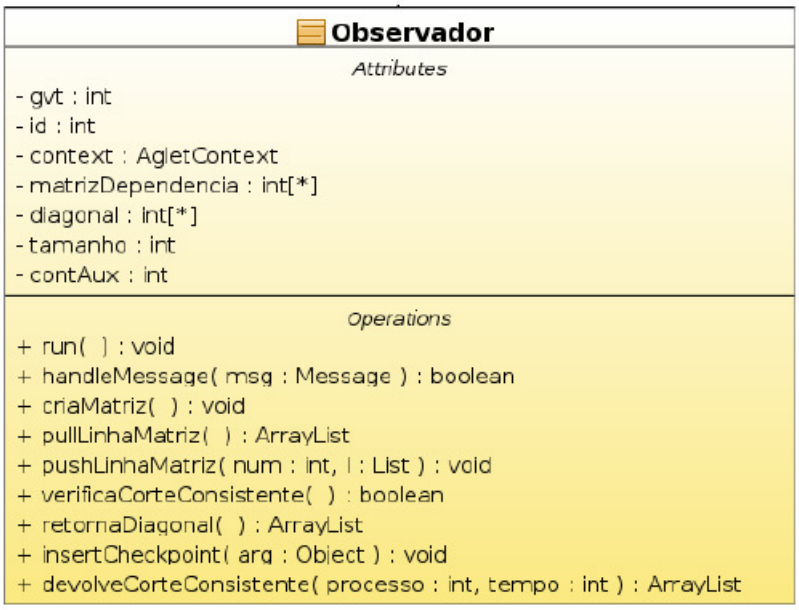

Figura 15: Diagrama da classe Observador.

O método ***run()*** é responsável por atualizar o contexto do agente observador. Após a migração para o seu refrente contexto de execução, o agente observador criar a matriz de dependência, inicializando-a com o valor padrão, ou seja, uma matriz identidade de ordem n onde n é a quantidade de processos em execução. Uma vez criada a matriz, toda a execução é baseada em troca de mensagens. O método ***handleMessage(...)*** é o responsável por receber e tratar as mensagens.

Ao se receber uma mensagem contendo um vetor de dependência (*checkpoint*), o processo Observador verifica se o vetor gera um corte consistente. Caso gere, a linha de recuperação é salva em um vetor de linhas de recuperação. Quando solicitada uma linha de retorno devido ao recebimento de uma mensagem fora da ordem de execução, o método ***devolveCorteConsistente(...)*** é invocado e retorna a linha de recuperação de menor custo para a simulação.

4.1.4 Tratamento de mensagens

Conforme exposto anteriormente, o tratamento dos eventos dentro da classe **Processo** não se dá dentro do método ***run()***. A razão para isto é não ter que refazer toda a simulação em caso de migração de contexto, devido à mobilidade fraca. A maneira utilizada para se contornar essa situação é fazendo com que um evento inicie o tratamento do próximo evento ao final da sua execução, por meio do envio de uma mensagem **TrataEvento** (Figura (**??**)). Caso essa abordagem não fosse adotada, todo o tratamento de mensagens seria bloqueado enquanto um processamento (laço *for*/*while*) estivesse em andamento.

Ao final do tratamento de um evento, a mensagem é enviada e enfileirada na fila de mensagens para ser tratada pelo processo ***handleMessage*(...)**. A mensagem em questão recebe o mesmo tratamento que as demais mensagem, e é tratada segundo a ordem em que é recebida. O tratamento das mensagens na ordem em que são recebidas, sem prioridades para algum tipo específico de mensagem, garante a intercalação de execução dos eventos e recebimento de mensagens com eventos externos, mensagens de *rollback* e mensagens de atualização de dados.

Um segundo tipo de mensagem bastante importante no andamento da simulação é a mensagem **InserirEvento**. Essa mensagem indica a inclusão de um novo evento na lista de eventos futuros do processo, e carrega consigo um objeto do tipo Evento que deve ser alocado na lista de eventos. É com base nos dados desse objeto Evento recebido que se identifica uma mensagem *straggler*, e se dispara otratamento de *rollback*.

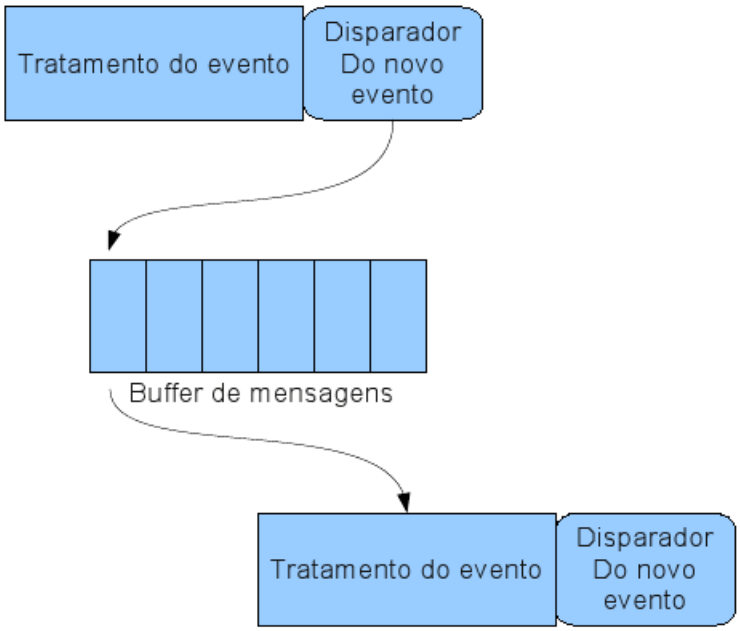

Figura 16: Ciclo de tratamento de eventos.

Além da mensagem de disparo para tratamento de um novo evento e da mensagem de recebimento de evento existem mensagens referentes à realização de *checkpoint* forçado, migração, realização de *rollback*, inicio do tratamento da lista de eventos, dentre outros. A lista completa de mensagens dos agentes **Processo** e **Observador** encontra-se no apêndice ??.

4.2 Considerações finais

A implementação deste trabalho compreende grande partes das ferramentas necessárias para a implementação de um simulação distribuída utilizando o protocolo *Rollback* Solidário sobre a biblioteca de agentes móveis *Aglets*. Foram desenvolvidas ferramentas para migração, detecção de mensagens recebidas fora da ordem (*straggler*), detecçao de linha de recuperação, em caso de *rollback*, além da estruturação necessária para se inserir um código de simulação distribuída sobre o protocolo.

5 Conclusões

A implementação do protocolo *Rollback* Solidário utilizando a biblioteca de agentes móveis *Aglets*, proposto nesse trabalho, trouxe algumas ferramentas para a inserção de um código de simulação distribuída sobre uma rede de computadores (sistema distribuído). A utilização de uma biblioteca de agentes móveis em Java foi útil na simplificação na comunicação e na migração dos processos.

Durante a execução deste trabalho, várias etapas individuais foram cumpridas. Em um primeiro momento, a preocupação foi a de se compreender de forma satisfatória o funcionamento de um sistema de simulação distribuído (simulação discreta, no contexto desse trabalho). Foram introduzidos os conceitos de simulação orientada a eventos, relógios lógicos, relógios vetoriais, *checkpoints*, precedência causal, mensagens e anti-mensagens, mensagens *straglers*, cortes consistentes, linhas de recuperação e, finalmente, o funcionamento do protocolo *Rollback* Solidário na sua forma semi-síncrona, com processo observador.

Um segundo passo foi o aprendizado da biblioteca *Aglets*. Nessa etapa foram desenvolvidas pequenas aplicações utilizando-se a biblioteca *Aglets* e o servidor de agentes *Tahiti*. O processo de aprendizado foi baseado no manual oficial da biblioteca(**??**). Foram desenvolvidas as técnicas de criação de *aglets* através do servidor, criação de *aglets* através de outros agentes já existentes, clonagem de agentes, migração, compreendimentos sobre o ciclo de vida de um *aglet*, envio, recebimento e tratamento de mensagens, dentre outros pontos.

5.1 Sobre a Implementação

No desenvolvimento do protocolo, um primeiro obstáculo encontrado foi a impossibilidade de se invocar métodos de outros agentes. Um agente se comunica com outro exclusivamente através de mensagens, mesmo estando em um mesmo contexto. Para se solucionar este problema, foi utilizada uma mensagem de disparo de método. Junto à essa mensagem, é enviada o parâmetro a ser tratado pelo método invocado, através do argumento arg da mensagem enviada. Com isso foi possível a invocação de métodos de outros agentes, estejam eles no mesmo contexto ou até mesmo em outro contexto.

Decorrente dessa solução de se enviar o parâmetro do método como argumento de uma mensagem, ocorreu a necessidade de se enviar por vezes objetos que nào suportavam serialização. Esses objetos não podem ser enviados como argumento de uma mensagem (é o caso do objeto *AgletID*). Para isso foi utilizado o método *toString()* em conjunto com o construtor *AgletID(String str)*, da classe *AgletID* . Com isso, o objeto *AgletID* foi, no agente de origem, convertido em uma *String*, enviado ao destino como *String* e, ao chegar no seu agente destino, foi recriado a partir desse argumento enviado como *String*.

Outros objetos, como o *AgletContext*, não suportam serialização. Isso serve, de certa forma, para que em uma migração de contexto, o abjeto *AgletContext* criado não venha a ser errôneamente utilizada. Uma forma contornar essa situação é a de apagar objeto *AgletContext* antes de uma migração e atulizá-lo assim que o objeto chega ao seu novo contexto.

Por fim, alguns métodos utilizados no manual estão marcados como *Deprecated*. Um desses métodos foi largamente utilizado nesse trabalho, o método *getAgletProxy(URL,AgletID)* da classe *AgletContext*. Esse método consiste em receber o ID de um agente e a URL do seu contexto e, com isso, gerar um *proxy* que possibilite a comunicação entre esses agentes. O problema criado por esse método é que, quando se é requisitado um proxy de um agente que não pertence a mesma URL,

durante a comunicação podem ocorrer falhas nos envios das mensagens. Para a solução desse problema foram testadas algumas alternativas:

- **Utilizar um método que realize a mesma tarefa:** Não foi encontrado em toda a documentação um método que possa substituir este método em questão, sendo essa alternativa descartada;
- **Serializar e enviar o proxy de comunicação:** O objeto *AgletProxy* não suporta serialização, o que impossibilitou essa solução;
- **Centralizar todo o tráfego para o processo observador:** Essa solução consiste em realizar a comunicação sempre entre o observador e um processo. Caso um processo precise enviar uma mensagem a outro, essa mensagem seria enviada ao observador que redirecionaria a mensagem ao processo em questão. Essa solução resolveria o problema de comunicação entre os processos, mas poderia representar um gargalo no sistema. Esse é um dos pontos em aberto desse trabalho.

5.2 Contribuições Desse Trabalho

Com a finalização da implementação proposta nesse trabalho, uma série de ferramentas foram concluídas para a inserção de um código de simulação distribuída sobre o protocolo *Rollback* Solidário.

Os serviços de criação e despacho dos processos para seus hosts de destinos foram concluídos, assim como os processos de retirada de eventos da lista de eventos para tratamento. A inserção de um evento na lista de eventos futuros em tempo de execução, assim como a sinalização de recebimento de uma mensagem *straggler* também foi concluída. Também estão finalizados os métodos para retorno à um tempo anterior da simulação, caso ocorra um *rollback*.

No processo observador, todo o processo de recebimento de *checkpoint*, detecção de corte consistente com base na matriz de dependência, eleição da linha de

retorno e envio das mensagens de *rollback* com a linha de recuperação para os processos está concluída.

5.3 Trabalhos Futuros

Com o fim da implementação desse trabalho, alguns novos pontos podem ser considerados para dar continuidade a esse projeto:

- Criação de um método eficiente de envio de mensagens *multicast* para os processos;
- Solução do problema de envio de mensagens para processos fora da mesma rede. A solução proposta de centralizar o tráfego de mensagens pode ser analizada como uma alternatica para o caso;