UNIVERSIDADE FEDERAL DE ITAJUBÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM
CIÊNCIA E TECNOLOGIA DA COMPUTAÇÃO

Projeto e Implementação dos Protocolos Otimistas Time Warp e Solidary Rollback para Simulação Distribuída

Márcio Emílio Cruz Vono de Azevedo

Dezembro de 2012
Itajubá - MG

UNIVERSIDADE FEDERAL DE ITAJUBÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM
CIÊNCIA E TECNOLOGIA DA COMPUTAÇÃO

Márcio Emílio Cruz Vono de Azevedo

Projeto e Implementação dos Protocolos Otimistas Time Warp e Solidary Rollback para Simulação Distribuída

Dissertação submetida ao Programa de Pós-graduação em Ciência e Tecnologia da Computação como parte dos requisitos para a obtenção do título de Mestre em Ciência e Tecnologia da Computação.

Orientador: Prof. Dr. Edmilson Marmo Moreira

Co-orientador: Prof. Dr. Otávio Augusto Salgado Carpinteiro

Área de Concentração: Sistemas de Computação

Dezembro de 2012
Itajubá - MG

Resumo

Este trabalho apresenta o projeto de implementação dos protocolos de sincronização para simulação distribuída intitulados *Time Warp* e *Solidary Rollback*. No desenvolvimento do projeto, um novo *framework* para o desenvolvimento de aplicações de simulação foi especificado e construído. Tanto o *framework* quanto a implementação realizada, foram elaborados considerando a plataforma Java, permitindo maior flexibilidade e extensibilidade. Para a validação das implementações realizadas, foram elaborados modelos de simulação para serem executados nos programas desenvolvidos com ambos os protocolos. Além disso, uma versão sequencial foi desenvolvida e serviu de referência para os resultados das simulações. Em adição, um estudo comparativo entre os dois protocolos é apresentado.

Palavras-chave: simulação distribuída, protocolos de sincronização, *Time Warp*, *Solidary Rollback*

Abstract

This paper presents the implementation project of the distributed simulation synchronization protocols named Time Warp and Solidary Rollback. Along the development o the project, a new framework for simulation application development was specified and built. Both, the framework and the implementation, were made considering the Java platform, allowing better flexibility and extensibility. In order to validate the implementations made, it was elaborated simulation models to be executed at the developed programs with both protocols. Furthermore, a sequential version was developed and used as a touchstone for the simulation results. In addition, a comparative study between both protocols were presented.

Keywords: distributed simulation, synchronization protocols, Time Warp, Solidary Rollback

“Embora mil vezes mil homens,
possa alguém em batalhas conquistar,
ainda assim, maior conquistador é
aquele que conquista a si mesmo.”
(Dhammapada - verso 103)

Dedico este trabalho ao meu pai, Oldemar Vono de Azevedo
(*in memoriam*).

Agradecimentos

Ao Buddha, ao Dhamma e à Sangha, que me serviram de refúgio nos momentos mais difíceis, trazendo serenidade.

À Deus e às forças espirituais superiores, por toda a ajuda recebida.

À minha namorada, Mara Ribeiro, pelo incentivo constante e dedicação a mim.

Aos meus pais, Lígia de Oliveira Cruz e Azevedo e Oldemar Vono de Azevedo (*in memoriam*), que me deram carinho e educação, moldando a pessoa que sou hoje.

Ao meu orientador, Prof. Dr. Edmilson Mármo Moreira, pela paciência e dedicação em diversos momentos da confecção deste trabalho.

À equipe do Inatel Competence Center, em especial ao Leonardo Liao, que autorizou minha ausência em preciosos momentos para a conclusão do presente trabalho.

À equipe da Ericsson TV de Denver, Colorado, em especial ao Kiran Ganuthula e Jamil Modak, que da mesma forma me permitiram ausentar todas as vezes que necessitei, sem questionamentos.

Ao curso de Sistemas de Informação da Universidade do Vale do Sapucaí, em especial ao Prof. José Luiz da Silva, pelo apoio e paciência.

À LIVEWARE, Tecnologia a Serviço Ltda., em especial ao meu ex-chefe Marcos Okita, por ter me liberado para fazer as disciplinas no início do curso.

Aos colegas do curso de Mestrado em Ciência e Tecnologia da Computação da UNIFEI, em especial ao amigo Roberto Ribeiro Rocha, pelo companheirismo que sempre demonstrou.

Aos amigos, em especial ao casal Crishna Irion e Thiago Borborema, por sempre me apoiarem.

A todos aqueles que, direta ou indiretamente, me ajudaram a chegar onde estou.

Sumário

1 Introdução **2**

2 Simulação **4**

2.1 Conceitos de Simulação 4

2.2 Sincronização em Simulação Distribuída 6

2.3 *Time Warp* 9

2.4 *Solidary Rollback* 12

2.4.1 *Checkpoints* Globais Consistentes 12

2.4.2 Descrição do *Solidary Rollback* 14

2.5 Considerações Finais 16

3 ***Framework*** **para o Desenvolvimento de Programas de Simulação Distribuída** **18**

3.1 Arquitetura em Camadas 18

3.2 Modelagem do *Framework* 19

3.3 Modelagem dos Protocolos Otimistas *Solidary Rollback* e *Time Warp* . . . 23

3.4 Considerações Finais 24

4 Adaptação e Descrição da Implementação do ***Framework*** **de Simulação Distribuída** **26**

4.1 Escolha das tecnologias 26

4.2 Contribuição ao modelo para implementação em Java 27

4.3 Aspectos práticos da implementação do *framework* 33

4.4 Considerações Finais 36

5 Experimentos Realizados e Discussão de Resultados 37

5.1 Modelos utilizados nas simulações 38

5.1.1 Modelos arbitrários 38

5.1.2 Modelos gerados aleatoriamente 39

5.2 Resultado das simulações 42

5.3 Considerações finais 44

6 Conclusão 45

6.1 Contribuições 45

6.2 Análise comparativa 46

6.3 Trabalhos futuros 46

A Descrição das *tags* do XML representativo de um modelo 50

Capítulo 1

Introdução

A simulação é uma ferramenta muito útil para solução de problemas do mundo real. Através de modelos simplificados, pode-se analisar o sistema real sem a necessidade de efetivamente implantá-lo. Banks (1999) afirma que um modelo deve ser complexo o suficiente para responder as questões levantadas, mas não tão complexo para que não se torne um problema tão grande quanto o real.

A simulação computacional é aquela que necessita de um computador para ser realizada. Um tipo de simulação computacional muito utilizado é a simulação de eventos discretos, onde os estados mudam discretamente no tempo. Dessa forma, as atividades na simulação de eventos discretos computacionais causam avanço no tempo.

Segundo Fujimoto (1999), existem vários motivos que levam a divisão da simulação computacional de eventos discretos em vários computadores. Entre eles, a redução do tempo de execução, a distribuição geográfica e a tolerância a falhas. A redução do tempo de execução da simulação se torna especialmente importante quando ela leva muito tempo para executar de forma sequencial.

O principal problema da simulação distribuída é a sincronização. Espera-se que o comportamento seja o mesmo de uma simulação sequencial, o que leva à necessidade de se observar a restrição de causalidade local. Os protocolos de sincronização conhecidos se dividem em duas categorias: protocolos conservativos e protocolos otimistas.

Os protocolos de sincronização conservativos asseguram que a simulação estará sempre sincronizada. Isso pode causar alguns problemas, como o aumento do tempo de simulação, pelo fato dos processos envolvidos estarem sempre esperando pelo momento seguro para executar o próximo evento, podendo até mesmo causar *deadlocks*. Técnicas para evitar tais problemas trazem consigo um *overhead* de mensagens de controle na comunicação entre os processos.

Os protocolos otimistas, ao contrário dos conservativos, permitem que a simulação

execute livremente, supondo, de forma otimista, que ela sempre se manterá sincronizada. Ao ocorrer uma falha na sincronização, devido a um processo receber uma mensagem que fira a restrição de causalidade local, os protocolos otimistas tomam atitudes para que seja realizado um *rollback* da simulação para um estado seguro para que a mensagem seja processado. A mensagem que fere a restrição é conhecida como mensagem *straggler*.

O protocolo *Time Warp*, proposto por Jefferson (1985), é o protocolo otimista mais conhecido. Ele utiliza o conceito de anti-mensagens no processo de *rollback*. Sempre que uma mensagem é enviada, uma anti-mensagem de sinal inverso é salva no processo remetente para que, em caso de *rollback*, elas possam ser desfeitas.

O protocolo *Solidary Rollback*, proposto por Moreira (2005), é uma alternativa ao *Time Warp* que não utiliza anti-mensagens no procedimento de *rollback*. Ao invés disso, todos os processos envolvidos na simulação voltam de forma colaborativa para um ponto comum e seguro para a mensagem *straggler* ser processada, chamado *Checkpoint* Global Consistente.

Cruz (2009) propôs um *framework* de simulação distribuída e apresentou a implementação dos protocolos *Time Warp* e *Solidary Rollback*. O *framework* proposto foi implementado em linguagem C++.

O presente trabalho tem por objetivo apresentar uma alternativa ao *framework* de Cruz (2009), através de uma implementação em Java. Tal alternativa foi inicialmente baseado no *framework* original, porém reformula as classes que manipulam os processos envolvidos na simulação, o protocolo de sincronização e o mecanismo de salvamento de estados.

O fato do novo *framework* ser implementado em tecnologia Java traz algumas vantagens estratégicas. Entre elas, a utilização de reflexão, que envolve, entre outros conceitos, o carregamento de classes em tempo de execução, além de permitir a agregação do amplo leque de tecnologias que orbitam a tecnologia Java.

Para alcançar o objetivo de apresentar um novo *framework*, fez-se necessário realizar um estudo de simulação distribuída de eventos discretos, dos protocolos otimistas de sincronização e do *framework* original apresentado por Cruz (2009). A validação da nova implementação foi realizada através da comparação dos resultados obtidos com uma implementação sequencial, utilizando três modelos diferentes.

Este trabalho está dividido em seis capítulos, incluindo a presente introdução. O segundo capítulo consiste de uma revisão bibliográfica sobre simulação. O terceiro capítulo realiza um estudo do *framework* apresentado por Cruz (2009). O quarto capítulo apresenta o desenvolvimento do novo *framework* proposto. O quinto capítulo apresenta os experimentos realizados para a validação da implementação do novo *framework* e o último capítulo apresenta as considerações finais do trabalho.

Capítulo 2

Simulação

Este capítulo tem por objetivo apresentar os conceitos básicos de simulação distribuída bem como apresentar os protocolos otimistas *TimeWarp* e *Solidary Rollback* que foram utilizados neste trabalho.

2.1 Conceitos de Simulação

Banks (1999) define simulação como sendo a imitação de um processo do mundo real sobre o tempo, envolvendo geração de histórico artificial do sistema e a realização de inferências sobre as características do sistema real. O autor complementa que a simulação é uma metodologia de solução de problemas do mundo real, onde se pode analisar o comportamento do sistema, formular questões sobre ele e ajudar no seu projeto. Através da simulação pode-se modelar sistemas existentes e conceituais.

Chwif e Medina (2007) classificam a simulação em computacional e não-computacional. A primeira necessita de um computador para ser realizada. A segunda não necessita de computador, e pode-se citar como exemplo um protótipo em escala reduzida de uma aeronave em um túnel de vento.

Para se realizar a simulação, é necessário construir um modelo do sistema a ser simulado. De acordo com Mello (2001), um modelo incorpora as características representativas do sistema real e são considerados uma descrição desse sistema. A execução do modelo em um computador potencializa resultados mais precisos sem a necessidade de intervenção no sistema real. A análise estatística dos resultados gerados pode contribuir na tomada de decisões.

Banks (1999) afirma que os modelos devem ser complexos o suficiente para responder as questões levantadas, mas não muito complexos. Dessa forma, Chwif e Medina (2007) complementam que o modelo é uma abstração do sistema real, aproximando de seu

comportamento, porém de forma mais simples. Se um modelo tiver uma complexidade maior do que o sistema real, ele passa a ser um problema, não um modelo.

Os modelos de simulação podem ser categorizados em discretos e contínuos. Os modelos discretos são aqueles em que os estados mudam discretamente no tempo. Já os modelos contínuos são aqueles em que seus estados mudam continuamente no tempo (NANCE; SARGENT, 2002).

Define-se simulação de eventos discretos como aquela que contém atividades que causam avanço no tempo. Banks (1999) afirma que a maioria das simulações de eventos discretos define intervalos como entidades de espera. Assim, os eventos consistem do início e fim de uma atividade ou de um intervalo. Nos modelos de simulação de eventos discretos, as variáveis que constituem o estado mudam somente em pontos discretos no tempo em que os eventos acontecem.

Ainda segundo Banks (1999), os eventos ocorrem como uma consequência dos tempos das atividades e dos intervalos. Entidades podem utilizar recursos do sistema, competindo por eles e entrando em filas de espera quando o recurso necessário não estiver imediatamente disponível. A "execução" do modelo de simulação de eventos discretos é realizada de tal forma que o tempo simulado é movido para frente. A atualização do estado do sistema é realizada em cada evento (tempo de início e fim de atividade ou intervalo), sendo que a captura e liberação de recursos podem ocorrer naquele tempo.

Também é possível separar a simulação de eventos discretos em diversos computadores. Nesse contexto, pode-se definir a simulação paralela e distribuída. Fujimoto (2000) define simulação paralela de eventos discretos referindo-se à execução dos programas responsáveis pela simulação em plataformas computacionais de múltiplos processadores. Quanto à simulação distribuída, o mesmo autor se refere à execução dos programas em computadores que se encontram distribuídos geograficamente, interligados por redes de computadores. Em ambos os casos a execução de um único modelo de simulação é distribuída em diversos computadores. Tal modelo pode ser composto por vários programas de simulação.

Existem várias razões para se dividir a simulação em diversos processadores. Fujimoto (1999) cita quatro: redução do tempo de execução, distribuição geográfica, execução do mesmo modelo em plataformas de diferentes fabricantes e tolerância à falhas.

Quando a computação da simulação é dividida em diversas sub-computações e essas são executas concorrentemente em diversos processadores, reduz-se o tempo de execução por um fator quase correspondente ao número de processadores que forem utilizados (o fator de redução só não é correspondente exatamente ao número de processos devido ao *overhead* intrínseco da simulação, da troca de mensagens entre o processos, entre outros fatores que causam *overhead*). Isso se torna potencialmente importante para simulações que levam muito tempo para executar, podendo requerer dias ou semanas para uma única

execução.

Sobre a distribuição geográfica da simulação, Fujimoto (1999) utiliza as seguintes palavras: “executando o programa de simulação em um conjunto de computadores geograficamente distribuídos é possível criar mundos virtuais com múltiplos participantes que estão fisicamente localizados em diferentes lugares. Isso alivia enormemente as despesas aéreas associadas à criação de exercícios conjuntos envolvendo participantes em diversas localizações”.

Também é possível executar a simulação em um ambiente interligando computadores de diferentes fabricantes. Fujimoto (1999) afirma que em certos casos pode-se ter um custo mais eficiente ao se interligar simuladores existentes, cada um desenvolvido para um computador específico, ao invés de se portar esses programas para um único simulador. Dessa forma, cria-se um novo ambiente virtual.

Utilizando-se múltiplos processadores pode-se desfrutar de outro benefício potencial que é a tolerância à falhas. Assim, em caso de falha de um processador, se o elemento crítico não residir no processador falho, torna-se possível continuar a simulação em outros processadores.

2.2 Sincronização em Simulação Distribuída

No contexto da simulação distribuída, a sincronização da simulação se torna uma grande área de pesquisa. A simulação distribuída deve apresentar os mesmos resultados da simulação sequencial, porém isso só é possível através de mecanismos de sincronização, preocupação que não existe na sequencial, pelo fato dos eventos ocorrerem naturalmente ordenados no tempo. Fujimoto (2000) afirma que esse é um problema central em simulação de eventos discretos na forma distribuída e tem atraído uma quantidade considerável de pesquisa.

A sincronização na simulação distribuída consiste do mecanismo responsável por garantir a reprodução correta dos relacionamentos “antes-e-depois” no sistema sendo simulado, algo que ocorre naturalmente na simulação sequencial. Fujimoto (2000) afirma que mecanismos de sincronização, conservativos e otimistas, foram produzidos para esse fim. Tais mecanismos tratam a simulação como uma coleção de processos lógicos que trocam mensagens (ou eventos) entre si. Cada mensagem carrega um *timestamp*, que é um valor correspondente ao tempo em que a mensagem deverá ser tratada pelo processo receptor. O objetivo é assegurar que seja respeitada a “restrição de causalidade local”, ou seja, assegurar que as mensagens sejam tratadas pelos processos lógicos de acordo com a ordem ditada pelo *timestamp* (mensagens com menor *timestamp* serão tratadas antes das mensagens com maior *timestamp*). Assim, a execução da simulação distribuída

terá o mesmo resultado da execução sequencial. Fujimoto (2000) ainda afirma que essa propriedade assegura que a simulação é repetível, ou seja, para os mesmos dados e parâmetros de entradas, os mesmos resultados são produzidos para cada execução.

Para melhor entendimento, faz-se necessário conhecer a construção básica de cada processo lógico envolvido na simulação. Para sua execução, cada processo possui um relógio local e uma lista de eventos futuros. As mensagens recebidas de outros processos são armazenadas na lista de eventos futuros na ordem de seu *timestamp*. Ao processar uma mensagem retirada da lista, a com menor *timestamp*, o processo avança seu relógio para o *timestamp* da mensagem. Para isso, é necessário assegurar, através de protocolos de sincronização, que o *timestamp* da mensagem a ser processada seja sempre maior do que o valor atual do relógio local. Assim, assegura-se que a restrição de causalidade local será sempre obedecida.

Uma vez que as mensagens são processadas na ordem do *timestamp*, pode-se estabelecer uma relação causal entre os eventos na simulação. Assim, se uma mensagem M1 é processada em um processo lógico LP1 e como resultado é gerada uma mensagem M2 que é enviada para o processo lógico LP2, através dos protocolos de sincronização assegura-se que a mensagem resultante M2 será processada em LP2 em um tempo lógico local maior do que o tempo quando a primeira mensagem M1 foi executada em LP1. Ou seja, se M2 tem uma dependência causal de M1, M2 é sempre processada em um tempo posterior a M1.

Afim de possibilitar a sincronização da execução dos eventos na simulação distribuída, conforme foi dito anteriormente, faz-se necessário a utilização de protocolos de sincronização. Tais protocolos podem ser classificados em "conservativos" e "otimistas". Os do primeiro tipo asseguram que os processos executem sempre sincronizados. Já os do segundo tipo, permitem que os processos executem livremente, porém, quando ocorre um evento que fere a restrição de causalidade local, a simulação é interrompida e ocorre um *rollback* para um ponto seguro para a mensagem ser processada. O termo em inglês *rollback* refere-se à volta do estado da simulação para um determinado ponto anterior no tempo. Assim, quando a simulação volta ao tempo em que estava sincronizada, o novo evento que ocorreu poderá ser executado sem ferir a restrição de causalidade local.

Os primeiros protocolos de sincronização propostos se basearam em uma abordagem conservativa (FUJIMOTO, 2000). Essa abordagem toma precauções que evitam a violação da restrição de causalidade local. Como exemplo, supondo que um determinado processo lógico encontra-se em um tempo local 10 e está pronto para processar uma mensagem com *timestamp* 15, se o processo tratar a mensagem imediatamente, pode acontecer que outro processo envie para ele uma mensagem com *timestamp* menor do que 15, por exemplo, 12. Então, o protocolo conservativo trava o processo até assegurar que os demais processos já estejam em um tempo local maior do que 15. Assim, Fujimoto (1999) complementa que a

principal função de um protocolo conservativo é determinar quando é "seguro" processar um evento.

Os primeiros algoritmos de sincronização foram os propostos por Bryant (1977) e Chandy e Misra (1978). Devido aos nomes dos autores, a combinação desses algoritmos ficou conhecida como CMB (abreviação de Chandy–Misra–Bryant). Para a execução do algoritmo, assume-se que: a informação sobre quais processos lógicos enviam mensagens para quais outros é fixa e conhecida; cada processo lógico envia mensagens com *timestamp* não decrescentes; a rede de comunicação assegura que as mensagens são recebidas na ordem que foram enviadas, ou seja, as mensagens que chegam em cada canal de entrada do processo lógico são recebidas na ordem ditada pelo *timestamp*.

Dadas as restrições acima, o processo lógico mantém uma fila de mensagens recebidas para cada canal de entrada ligado a cada outro processo lógico que envia mensagens para ele. A fila que armazena essas mensagens é do tipo FIFO (do inglês *first-in-first-out* – o primeiro a chegar é o primeiro a sair). Como as mensagens chegam na ordem do *timestamp*, então cada fila de entrada é ordenada pelo *timestamp*. As mensagens direcionadas do processo lógico para ele mesmo são também inseridas em uma fila de mensagens. Assim, tem-se uma fila para cada canal de entrada e mais uma para mensagens direcionadas para o próprio processo. Para cada fila é associado um relógio lógico correspondente ao *timestamp* da primeira mensagem da fila, ou, no caso da fila estar vazia, da última mensagem processada. O processo, então, tratará sempre a primeira mensagem da fila que tem o menor relógio lógico. Se essa fila não contiver mensagem, então o processo é bloqueado até que chegue uma mensagem para o canal associado à fila. Assim os eventos são processados na ordem do *timestamp*.

Caso exista uma cadeia circular de processos em que cada um possa enviar mensagem para o próximo e todos possam ficar com a fila de entrada vazia associada ao canal correspondente à cadeia circular, ocorre um *deadlock*, pois cada processo ficará esperando mensagens do outro adjacente. Fujimoto (1999) afirma que se existir um número relativamente pequeno de mensagens de eventos não processados comparado com o número de ligações na rede, ou se os eventos não processados começarem a *clusterizar* em uma parte da rede, ou seja, permaneçam em um subconjunto dos processos envolvidos na simulação, os *deadlocks* podem ocorrer com frequência.

Para resolver o problema dos *deadlocks*, um método utilizado é o envio de mensagens nulas. Sempre que um processo trata um evento, ele envia uma mensagem nula para cada ligação de saída com o *timestamp* atual do processo. Assim, pode-se dizer que com o envio da mensagem nula o processo que a enviou está prometendo que nenhuma outra mensagem chegará naquele canal com um *timestamp* menor do que o *timestamp* da mensagem nula. Entretanto, Bui, Lang e Workman (2003) afirmam que muitas mensagens nulas podem ser geradas, degradando o desempenho da simulação.

Moreira (2005) acrescenta que uma variação ao mecanismo de mensagens nulas citado acima consiste em enviar tais mensagens nulas sob demanda, e não após o processamento de cada evento. A frequência dessa demanda pode ser dada por um temporizador ou quando o menor relógio de todos os canais for o de uma fila vazia.

Além da abordagem conservativa, existe também a abordagem otimista que, conforme foi dito anteriormente, permite que a simulação execute livremente até a chegada de uma mensagem que fira a restrição de causalidade local, quando é realizado um *rollback* da simulação para um ponto em que seja seguro processar tal mensagem sem ferir a restrição mencionada. O mais conhecido desses protocolos é o chamado *Time Warp*. Moreira (2005) também propõe um outro protocolo otimista chamado *Solidary Rollback*, ou *Rollback* Solidário. Ambos os protocolos serão estudados nas próximas seções, pois os mesmos foram utilizados na confecção desse trabalho.

2.3 *Time Warp*

O *Time Warp* é um protocolo otimista para sincronização de simulação distribuída. Malik, Park e Fujimoto (2009) afirmam que ele é o método mais bem conhecido para tal fim. Os conceitos fundamentais usados pelos protocolos otimistas, como *rollback* e controle global do tempo, foram descritos inicialmente no *Time Warp*. Este protocolo foi apresentado por Jefferson (1985) e será descrito a seguir de acordo com este artigo.

Jefferson (1985) propõe o *Virtual Time* como sendo um novo paradigma para organizar e sincronizar sistemas distribuídos. No mesmo artigo, é proposta uma implementação para o *Virtual Time* que é o protocolo *Time Warp*.

Para a execução do *Time Warp*, assume-se que qualquer processo é livre para enviar mensagens para outros processos, inclusive ele mesmo. Assim, não há o conceito de canal entre dois processos, não havendo, portanto, a necessidade de protocolos de abertura e fechamento desses canais.

As mensagens trocadas entre os processos possuem, além dos dados transportados, os seguintes campos: nome ou identificador do processo que enviou; tempo virtual de envio, ou seja, o relógio lógico do processo emissor no momento do envio; nome ou identificador do processo que irá receber a mensagem e tempo virtual de recebimento, ou seja, o tempo virtual no processo destinatário em que a mensagem deverá ser processada.

Algumas regras devem ser obedecidas no tratamento de eventos e no preenchimento dos campos das mensagens. Segundo Jefferson (1985), estas regras são: (1) o tempo virtual de envio de cada mensagem deve ser menor do que o tempo virtual de recebimento; (2) o tempo virtual de cada evento em um processo deve ser menor do que o tempo virtual

do próximo evento naquele processo. Tais regras são exatamente o que é conhecido como *Lamport's Clock Conditions*, definido por Lamport (1978).

Lamport (1978) define a relação chamada "*happens before*" (expressão em inglês que significa "acontece antes"), descrevendo a precedência causal entre os eventos. Tal relação foi então formalizada da seguinte forma: (1) se um evento A causa um evento B, então a execução de A e B devem ser agendadas no tempo real de tal forma que A estará completo antes de B iniciar; (2) se A e B tem a mesma coordenada de tempo virtual, então não há restrição em seus agendamentos, e eles podem ser realizados arbitrariamente. Assim, Jefferson (1985) afirma que os tempos virtuais no *Time Warp* são somente parcialmente ordenados, apesar de seu trabalho assumir que eles são totalmente ordenados.

Jefferson (1985) considera que o *timestamp* da mensagem é, na verdade, o tempo virtual de recebimento. Então, doravante, tal tempo virtual será referenciado como *timestamp*.

Para o *Time Warp*, é necessário e suficiente que cada mensagem do processo seja manipulada na ordem do *timestamp*. Porém, as mensagens geralmente não chegarão ao processo na ordem do *timestamp*.

O protocolo *Time Warp* possui duas partes principais: o mecanismo de controle local e o mecanismo de controle global. O primeiro trata de assegurar que os eventos sejam executados e as mensagens sejam processadas na ordem do *timestamp*. O segundo trata de problemas globais, como coleta de lixo, utilizando-se do cálculo do tempo virtual global (GVT).

Apesar do tempo virtual das mensagens ser global, cada processo tem o seu relógio virtual local. A qualquer momento, alguns relógios virtuais podem estar a frente de outros, porém isso é invisível para cada processo, pois eles somente têm acesso a seus relógios locais.

As mensagens recebidas por um determinado processo são armazenadas em uma fila ordenada pelos *timestamps*. Idealmente, imagina-se, de forma otimista, que nenhuma mensagem chegará ao processo com um *timestamp* anterior ao relógio lógico do processo. Apesar de isso poder ocorrer, tal situação será tratada mais adiante.

A estrutura interna de um processo possui campos que possibilitam a execução do *Time Warp*. O **nome do processo** dever ser um identificador único para ele. O **tempo virtual local**, abreviado como LVT (do inglês, *local virtual time*), nada mais é do que o relógio virtual local do processo. O **estado** é o conjunto de variáveis que compõem o estado atual do processo. A **"pilha" de estados** é a estrutura onde as cópias dos estados são armazenadas de tempos em tempos, para serem recuperadas em uma eventual necessidade de *rollback*. A **fila de entrada** é a estrutura onde as mensagens são armazenadas após chegarem ao processo. Tal fila é ordenada pelo *timestamp* das mensagens. Mesmo após

processadas, as mensagens continuam na fila para serem "desfeitas" em caso de *rollback*. A **fila de saída** contém uma cópia negativa de cada mensagem enviada pelo processo para outros processos, utilizadas em caso de *rollback* para desfazer mensagens já enviadas para outros processos. Tais mensagens são armazenadas pelo tempo virtual de envio e são chamadas de "antimensagens" (JEFFERSON, 1985).

Para cada mensagem existente no sistema existe uma antimensagem. A antimensagem é exatamente igual a sua mensagem equivalente, porém com sinal contrário. A mensagem possui sinal positivo e a antimensagem negativo. Quando um processo cria uma mensagem e envia para outro processo, ele também cria e armazena uma antimensagem correspondente à mensagem original. Uma vez que um par mensagem / antimensagem se encontre novamente na fila de um mesmo processo, elas são mutuamente aniquiladas, independente de qual mensagem chegou primeiro na fila.

Uma mensagem normalmente chega na fila de entrada de um processo com um *timestamp* maior do que o LVT do processo. Nesse caso, a mensagem é simplesmente inserida na fila de entrada. Porém, pode ocorrer que uma mensagem chegue com *timestamp* menor do que o LVT, ocasionando então um *rollback* para um estado cujo LVT seja menor do que o *timestamp* da mensagem, quando então a mensagem é inserida na fila de entrada e processada.

O procedimento de *rollback* procura, na pilha de estados, um estado salvo com LVT anterior ao *timestamp* da mensagem. O estado é restaurado e a mensagem poderá ser inserida na fila de eventos futuros. No entanto, o processo pode ter enviado mensagens entre o LVT original, antes da chegada da mensagem *straggler* (mensagem com *timestamp* anterior ao LVT), e o LVT para o qual o estado foi restaurado. Então, todas aquelas mensagens devem ser desfeitas. Para isso, o processo retira da fila de saída todas as antimensagens correspondentes às mensagens enviadas naquele intervalo e envia para os destinatários da mensagem original, informação também encontrada na antimensagem.

Quando um processo destinatário recebe uma antimensagem, três tipos de ações são tomadas, dependendo do *timestamp* da mensagem, do LVT do processo e do estado da fila de entrada no momento em que ela chega. Quando a mensagem original correspondente à antimensagem já estava na fila e ainda não tinha sido processada, a mensagem e a antimensagem aniquilarão uma à outra, sendo descartadas da fila. Quando a mensagem original positiva já tiver sido processada, deverá ser realizado um *rollback* para um estado com LVT anterior ao qual a mensagem tenha sido processada e novamente ambas se aniquilam e são descartadas da fila. Nesse caso, antimensagens também poderão ser geradas como resultado deste novo *rollback*. Quando a mensagem positiva ainda não tiver chegada no momento em que se recebe a antimensagem, devido à latência na rede, a antimensagem é inserida na fila e processada da mesma forma que uma mensagem positiva. Assim, quando a mensagem original chegar, ambas serão aniquiladas, e eventuais *rollbacks*

poderão ocorrer.

Jefferson (1985) afirma que o protocolo de antimensagem é extremamente robusto. Acrescenta que o processo de *rollback* deve ser atômico e que não há possibilidade de "efeito dominó", pois no pior caso o *rollback* irá restaurar o estado inicial do processamento.

O *Time Warp* ainda possui, conforme foi dito anteriormente, um mecanismo de controle global, usado, por exemplo, para realização de coleta de lixo. O ponto central de tal mecanismo é o cálculo do tempo virtual global, ou GVT (do inglês, *Global Virtual Time*).

Jefferson (1985) define o GVT no tempo real r como o mínimo entre (1) todos os tempos virtuais de todos os relógios virtuais no tempo r e (2) os tempos virtuais de envio de todas as mensagens que foram enviadas mas ainda não foram ainda processadas no tempo r. Dessa forma, o GVT serve como piso para os tempos virtuais para os quais um processo pode realizar um *rollback*, sendo que qualquer informação anterior a esse tempo possa ser descartada.

A definição de GVT anterior não é totalmente operacional, levando-se em conta a impossibilidade de se determinar o LVT exato no tempo real r em cada processo. Um algoritmo para cálculo de GVT para sistemas distribuídos que endereça problemas como mensagens em trânsito foi proposto por Samadi (1985).

Segundo Moreira (2005), existem várias implementações conhecidas do *Time Warp*. Todas as implementações utilizam o mecanismo de antimensagens. Porém, Moreira (2005) propôs um novo protocolo otimista de sincronização de simulação distribuída de eventos discretos chamado *Solidary Rollback*. Tal protocolo será discutido na próxima seção.

2.4 *Solidary Rollback*

O *Solidary Rollback* é um protocolo otimista que pode ser considerado uma alternativa ao *Time Warp*, pois não utiliza antimensagens em sua implementação. Ao invés disso, ele utiliza o conceito de *Checkpoints* Globais Consistentes (MOREIRA; SANTANA; SANTANA, 2005) afim de identificar os pontos da simulação para onde o processo deverá realizar o *rollback*. Assim, faz-se necessário apresentar os conceitos envolvidos na identificação dos *Checkpoints* Globais Consistentes antes de apresentar efetivamente este protocolo.

2.4.1 *Checkpoints* Globais Consistentes

Manivannan e Singhal (1999) definem um *checkpoint* local como um estado intermediário salvo de um processo. Também definem um *checkpoint* global consistente como sendo um conjunto de *checkpoints* locais, um de cada processo envolvido na computação

distribuída, e não possuindo nenhuma dependência causal entre eles. Porém, Netzer e Xu (1995) afirmam que essa restrição de não dependência causal entre dois *checkpoints*, apesar de ser necessária para que eles façam parte de uma fotografia global consistente do sistema, não é suficiente. Por exemplo, na Figura 2.1 não existe dependência causal entre $C_{1,1}$ e $C_{3,1}$, mas eles não podem fazer parte de nenhum *checkpoint* global consistente, visto que não podem ser combinados com nenhum outro *checkpoint* $C_{2,x}$ do processo 2.

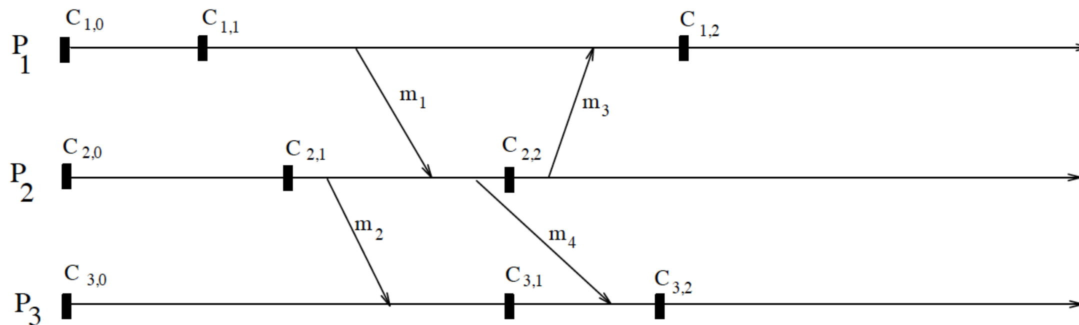

Figura 2.1: Caminhos Z não causais (MANIVANNAN; SINGHAL, 1999)

Uma vez definida qual é a condição necessária para dois *checkpoints* de processos distintos fazerem parte de um *checkpoint* global consistente, Netzer e Xu (1995) também definiram a condição suficiente para que isso aconteça como sendo a não ocorrência de um caminho Z (do inglês *Z-path* ou *zigzag path*) entre os dois *checkpoints* locais.

Define-se a existência de um caminho Z do *checkpoint* $C_{p,i}$ para o *checkpoint* $C_{q,j}$ se e somente se existir uma sequência de mensagens $m_1, m_2, ..., m_n$ (para $n \geq 1$) de tal forma que: (1) m_1 é enviado pelo processo p depois de $C_{p,i}$, (2) m_k (onde $1 \leq k \leq n$) é recebida pelo processo r e, no mesmo intervalo de checkpoint ou em um intervalo posterior, m_{k+1} é enviada pelo processo r, podendo ser enviada antes ou depois do recebimento de m_k, e m_n é recebida pelo processo q antes de $C_{q,j}$ (NETZER; XU, 1995).

No exemplo apresentado na Figura 2.1, o que impede que se forme um *checkpoint* global consistente entre os *checkpoints* $C_{1,1}$ e $C_{3,1}$ é o caminho Z formado pelas mensagens m_1 e m_2. Se existisse um *checkpoint* no processo P_2 entre o envio de m_2 e antes do recebimento de m_1, eliminando o caminho Z, seria possível ter um *checkpoint* global consistente adicionando os *checkpoints* anteriormente mencionados.

Existem vários algoritmos utilizados para prevenir a ocorrência de caminhos Z na execução, forçando a ocorrência de novos *checkpoints* locais, além daqueles que ocorreriam naturalmente nos momentos de salvamento de estados. Os algoritmos assíncronos são aqueles que marcam os *checkpoints* localmente, independente dos outros processos envolvidos na computação, em contraposição aos algoritmos síncronos, onde os processos sincronizam seus *checkpoints* globalmente. Os algoritmos assíncronos tem a desvantagem de possibilitar os caminhos Z e os síncronos tem a desvantagem de ter que interromper

a execução para sincronizar os processos para se obter um *checkpoint* global consistente. Então, define-se os algoritmos semi-síncronos como aqueles em que forçam as marcações para evitar os caminhos Z, resolvendo o problema dos algoritmos assíncronos de forma mais otimizada que os síncronos (MANIVANNAN; SINGHAL, 1999).

Entre os algoritmos semi-síncronos, está o FDAS (*Fixed Dependency After Send*), proposto por Wang (1997) e utilizado no *Solidary Rollback*. No FDAS, quando um processo envia uma mensagem, ela carrega consigo o vetor de dependências do processo. O componente do vetor de dependências correspondente ao processo atual é sempre incrementado após a marcação de um *checkpoint*, forçado ou não. Após enviar uma mensagem, se o processo receber uma outra mensagem, antes do próximo *checkpoint*, que carregue em seu vetor de dependências um componente maior do que o componente correspondente no vetor de dependências do processo, um *checkpoint* forçado é marcado, o relógio vetorial do processo é atualizado e então a mensagem é processada. Analisando o algoritmo, é possível comprovar que ele é capaz de evitar caminhos Z na execução.

Afim de compreender melhor o funcionamento do vetor de dependências, faz-se necessário ressaltar que ele é atualizado sempre que uma mensagem é recebida com as componentes dos outros processos, quando essas componentes do vetor vindo com a mensagem são maiores do que as componentes do vetor de dependências atual do processo.

2.4.2 Descrição do *Solidary Rollback*

No *Solidary Rollback*, além dos processos envolvidos na simulação, existe um processo observador, capaz de calcular os *checkpoints* globais consistentes durante a execução da simulação. Tais *checkpoints* globais consistentes detectados pelo observador são chamados de "linhas de recuperação", pois representam o ponto da simulação para onde todos os processos deverão retornar "solidariamente" no caso de ocorrer uma violação da restrição de causalidade local.

Afim de compor as linhas de recuperação, o processo observador monta uma matriz quadrada M de ordem n, onde n é o número de processos envolvidos na simulação (excluindo-se o observador). Ao iniciar sua execução, cada processo envia para o observador o seu vetor de dependências inicial, onde todos os componentes possuem valor 0, exceto o componente referente ao processo atual, que possui valor inicial 1. Os vetores correspondentes aos *checkpoints* iniciais são enviados para o observador. Cada vez que um processo marca um *checkpoint*, ele envia para o observador o vetor de dependências após a marcação, cujo componente correspondente ao processo é incrementado em 1, e os componentes correspondentes aos demais processos são atualizados conforme descrito anteriormente. Sempre que o observador recebe os vetores de um processo, ele atualiza a linha da matriz com o vetor correspondente ao processo que enviou.

Exemplificando o que foi dito, imagine que o processo observador, cuja matriz M no início da execução corresponde à apresentada na Equação 2.1, receba um vetor de dependências do processo P_2 cujo valor é $\begin{pmatrix}1 & 3 & 1 & 0\end{pmatrix}$.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.1}$$

O observador atualizará, então, sua matriz M para aquela mostrada na Equação 2.2.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 3 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.2}$$

O processo observador irá considerar uma linha de recuperação válida sempre que as componentes da diagonal da matriz forem maiores do que as componentes das colunas correspondentes. Por exemplo, a Equação 2.2 não apresenta uma linha de recuperação válida, pois as componentes da diagonal nas linhas 1 e 3 não são as maiores das colunas. Adicionalmente, pode-se demonstrar que quando as componentes da diagonal forem maiores do que as demais componentes de cada coluna, não existe dependência causal entre os *checkpoints* representados pelos vetores que compõem as linhas da matriz, assim sendo, tais *checkpoints* locais podem fazer parte do mesmo *checkpoint* global consistente.

Assim como no *Time Warp*, cada processo envolvido na simulação no *Solidary Rollback* possui um única fila de entrada onde as mensagens futuras são temporariamente armazenadas na ordem do *timestamp*, até que sejam processadas. Quando chegar uma mensagem *straggler*, é necessário realizar o *rollback* para um estado onde é seguro processar a mensagem. Assim, um conjunto de mensagens é trocado entre o processo atual e o observador e entre o observador e os demais processos de forma a retornar toda a simulação para uma linha de restauração definida pelo observador. Um esboço da execução do *rollback* com o auxílio do observador pode ser visto na Figura 2.2.

A Figura 2.2 mostra que o processo que recebe a mensagem *straggler*, aqui chamado de P_i, após detectar o estado para o qual o processo voltará, inicia o procedimento de *rollback* enviando uma mensagem notificando o processo observador, denominado como P_0. Essa notificação carrega consigo o componente do vetor de dependências correspondente ao estado para o qual processo voltará. Após enviar a notificação, P_i recupera o estado identificado e permanece aguardando a confirmação do observador. O observador, por sua vez, ao receber a notificação de *rollback*, identifica a linha de recuperação contendo

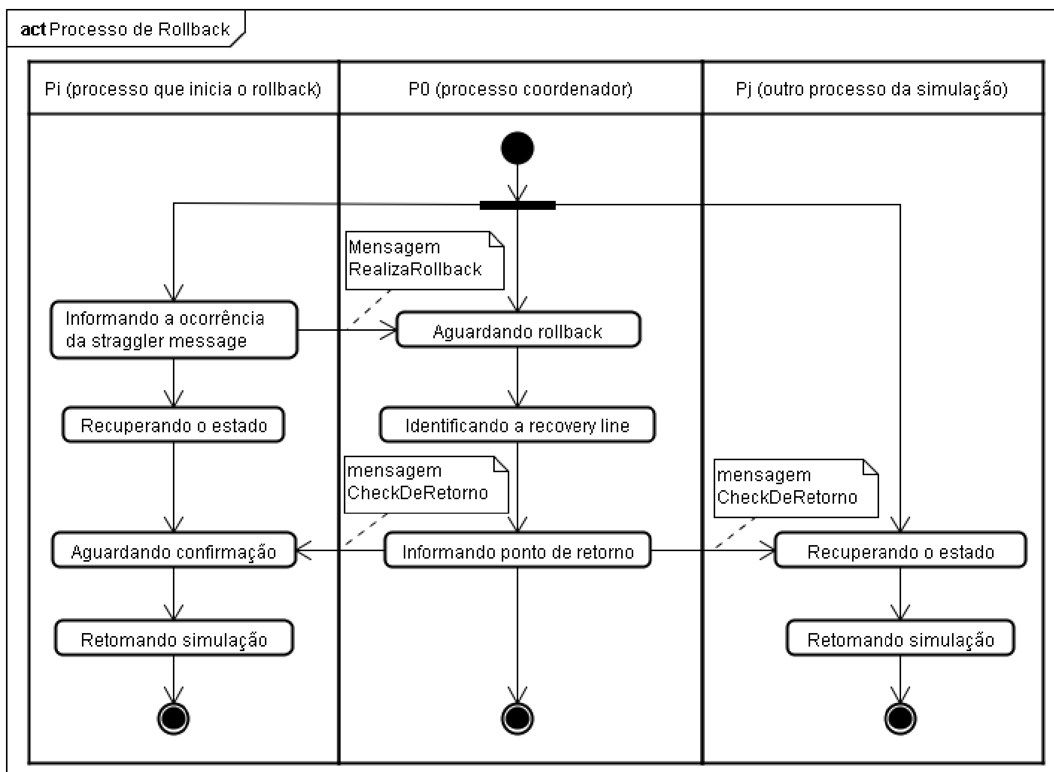

Figura 2.2: *Rollback* coordenado pelo observador no *Solidary Rollback* (adaptado de Moreira (2005))

o componente recebido do processo P_i. Então ele notifica o ponto de recuperação para todos os processos que participam da simulação. Quando P_i recebe a confirmação, ele simplesmente retoma a simulação. Quando os demais processos recebem a notificação com o ponto de recuperação, eles recuperam o estado pelo componente identificado e retomam a simulação. Os demais processos da simulação são identificados no diagrama como P_j. Devido ao fato de todos os processos retornarem a um estado salvo ao mesmo tempo, coordenados pelo observador, elimina-se a possibilidade de *rollbacks* em cascata e diminui-se o *overhead* de processamento observado no *Time Warp* devido à utilização de antimensagens.

O cálculo do GVT no *Solidary Rollback* torna-se bastante simplificado devido à utilização do processo observador, pois ele recebe constantemente os vetores de dependências e os LVTs de todos os processos envolvidos na simulação. Baseado em ambas as informações o GVT é calculado e sempre que ocorrer um *rollback* o processo observador pode aproveitar a oportunidade e enviar o GVT para todos os demais processos para que eles possam realizar a coleta de lixo.

2.5 Considerações Finais

Uma vez apresentadas as noções básicas de simulação distribuída e dos protocolos de sincronização, juntamente com os protocolos otimistas que foram utilizados neste trabalho, obteve-se o fundamento necessário para entender como aplicar tais conceitos em um *framework* de simulação distribuída. O próximo capítulo apresentará o *framework* proposto por Cruz (2009) que será aqui implementado utilizando ambos os protocolos apresentados.

Capítulo 3

Framework para o Desenvolvimento de Programas de Simulação Distribuída

Este capítulo se baseia no trabalho de Cruz (2009) que define um *framework* para o desenvolvimento de aplicações de simulação distribuída. Parte dos diagramas apresentados aqui seguem a notação baseada na UML (Linguagem de Modelagem Unificada, do inglês *Unified Modeling Language*), que é uma linguagem padrão para a elaboração da estrutura de projeto de *software* orientado a objetos (BOOCH; RUMBAUGH; JACOBSON, 2005).

3.1 Arquitetura em Camadas

Cruz (2009) define uma arquitetura em camadas para um ambiente de simulação distribuída conforme ilustra a Figura 3.1. São apresentadas, então, as camadas de **Aplicação**, **Sincronização**, **Comunicação** e **Arquitetura**.

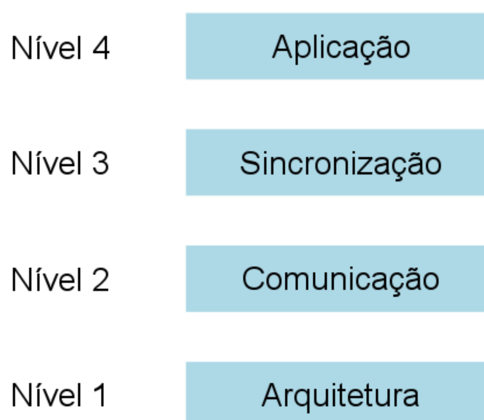

Figura 3.1: Arquitetura em Camadas para Simulação Distribuída (CRUZ, 2009)

A camada de **Arquitetura** é responsável por manter as informações da arquitetura

física do ambiente onde o programa de simulação será executado, por exemplo, o número de processadores existentes e a quantidade de memória alocada.

A camada de **Comunicação** é responsável pela troca de informações entre os processos durante a execução da simulação. Esta camada proverá a comunicação através de protocolos de troca de mensagens e manterá o canal de comunicação entre os processos envolvidos na simulação e possíveis processos controladores. Ela também é responsável por garantir que toda mensagem será recebida pelo processo receptor e que a ordem cronológica do envio será respeitada no recebimento. Exemplos de protocolos de comunicação que podem ser usados por esta camada são o PVM (Máquina Virtual Paralela, do inglês *Parallel Virtual Machine*), cujos detalhes podem ser encontrados em Geist et al. (1994), e o MPI (Interface de Passagem de Mensagem, do inglês *Message Passing Interface*), cujos detalhes podem ser encontrados em Walker (1994).

A camada de **Sincronização** é responsável por garantir a consistência dos resultados obtidos pela simulação, tratando, assim, os possíveis erros de causa e efeito e garantindo que a restrição de causalidade local de cada processo será respeitada. Dessa forma, a simulação distribuída obterá os mesmos resultados de uma simulação sequencial, exceto pela velocidade que se ganhará na obtenção dos resultados devido à paralelização. Exemplos de protocolos da camada de sincronização são o *Time Warp* (JEFFERSON, 1985) e o *Solidary Rollback* (MOREIRA, 2005), ambos detalhados no capítulo anterior.

Finalmente, a camada de **Aplicação** é onde os modelos de simulação, providos como entrada para o *framework*, serão efetivamente executados e onde os dados serão coletados e entregues em uma saída definida.

3.2 Modelagem do *Framework*

Cruz (2009) representou o comportamento estático do *framework* proposto através do diagrama de classes da UML. A Figura 3.2 apresenta tal diagrama.

A classe `Ambiente` define o *container* que hospeda os principais objetos do sistema. Ela possui uma associação "um para um" com cada uma das classes que representam tais objetos: `Arquitetura`, `Comunicacao`, `Protocolo` e `EstadoGeral`. As classes `Comunicacao`, `Protocolo` e `EstadoGeral` são classes abstratas, permitindo assim flexibilidade de implementação para cada um desses itens.

A classe `Ambiente` possui dois métodos principais: `PrepararAmbiente()` e `Executar()`. O primeiro inicializa as classes `Arquitetura` e `Protocolo`, sendo que este último aciona a preparação da classe `Modelo` que escalona os processos lógicos aos processadores conforme definido na classe `Arquitetura`. Esse escalonamento pode ser realizado de forma auto-

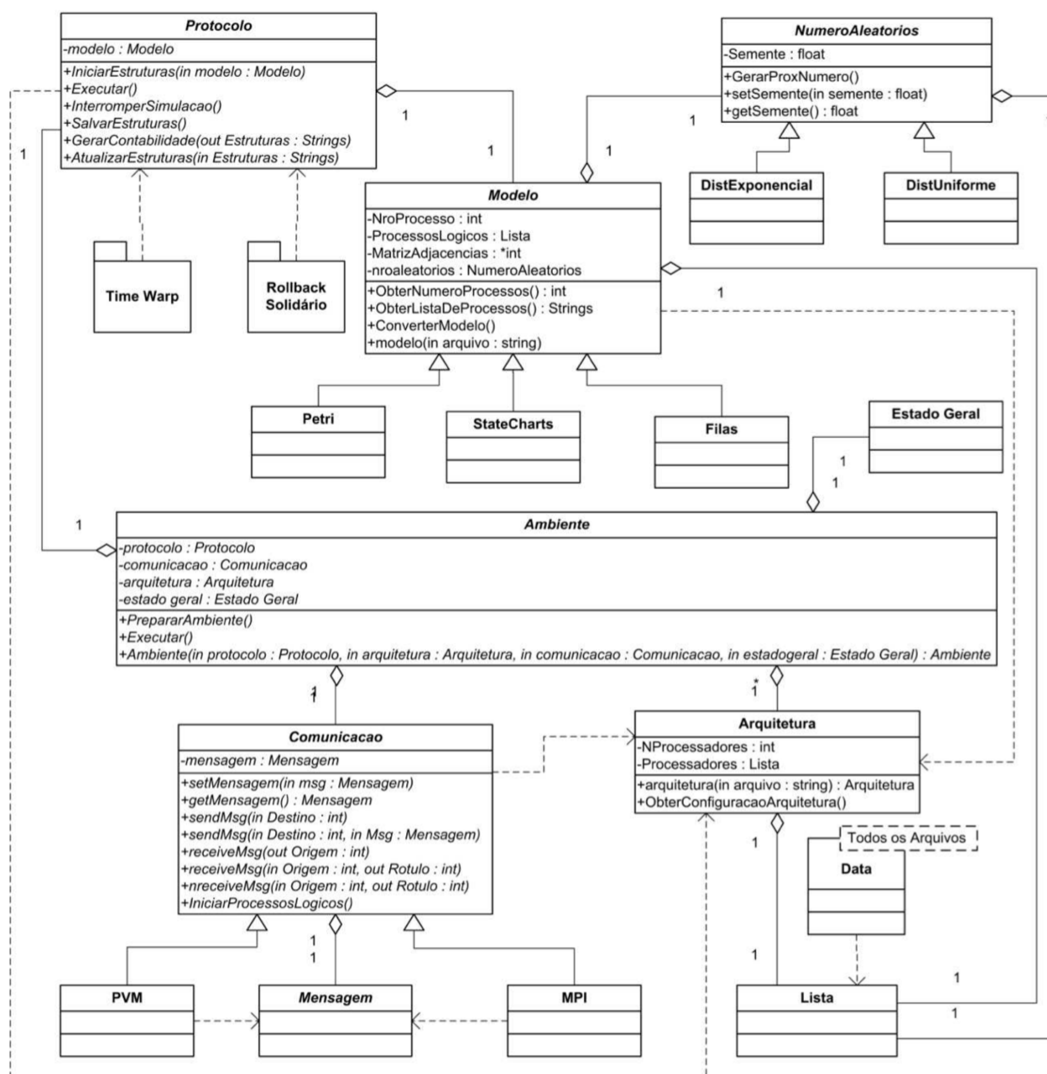

Figura 3.2: Diagrama de classes do *framework* proposto por Cruz (2009), extraído do mesmo trabalho

mática em protocolos de comunicação como o MPI, que foi adotado na implementação conforme será visto no Capítulo 4. É importante destacar que quando se trata da utilização das classes abstratas descritas, os construtores chamados correspondem às subclasses concretas de cada uma delas.

A classe `EstadoGeral` armazena as informações gerais da simulação e seu comportamento. É através dela que o usuário (programador da simulação) poderá especificar as informações que deseja analisar com base no modelo.

A classe `Arquitetura` é responsável pela definição de toda a arquitetura física do sistema onde a simulação será executada. Tal definição inclui informações como número de processadores, recursos de processamento, quantidade de memória entre outras. Cruz (2009) ainda apresenta um arquivo contendo os dados da arquitetura que, uma vez carregado, constrói o objeto da classe `Arquitetura` com estes dados. A Listagem 3.1

apresenta um exemplo de arquivo com dados baseados em um laboratório da Universidade Federal de Itajubá.

```
** Arquivo: Arquitetura.dat **

[Elementos]
Quantidade: 32

[Estacoes]
Quantidade: 32
Arquitetura: Intel Quad Core Q6600
Processador: 2,40GHz
Memoria: 2GB
Cash: 8MB
Disco: 250GB

Quantidade: 2
Arquitetura: Intel Xeon Dual Zion 5410
Processador: 2,33GHz
Memoria: *GB
Cash: 12MB
Disco: 4,5TB
```

Listagem 3.1: Exemplo de arquivo de arquitetura (adaptado de Cruz (2009))

Outra classe presente no diagrama é a classe `Modelo`, também abstrata, afim de permitir a geração dos modelos através de ferramentas de modelagem. Por exemplo, pode-se necessitar converter modelos baseados em redes de filas, *statecharts* ou redes de Petri. O método abstrato `ConverterModelo()` deverá ser implementado para realizar a conversão. Os dados que serão representados pelo objeto da classe `Modelo` são importados no sistema através de um arquivo de configuração, conforme mostrado na Listagem 3.2, cujo grafo representativo se encontra na Figura 3.3.

```
** Arquivo: Modelo.dat **

[Processos]
Quantidade: 3

[Processo 01]
DistChegada: Exponencial
TaxaNascimento: 50
TaxaMorte: 30
TaxaComunicacao: 0 20 50
```

```
[Processo 02]
DistChegada: Exponencial
TaxaNascimento: 0
TaxaMorte: 60
TaxaComunicacao: 20 0 20

[Processo 03]
DistChegada: Exponencial
TaxaNascimento: 0
TaxaMorte: 70
TaxaComunicacao: 30 0 0
```

Listagem 3.2: Exemplo de arquivo de modelo (adaptado de Cruz (2009))

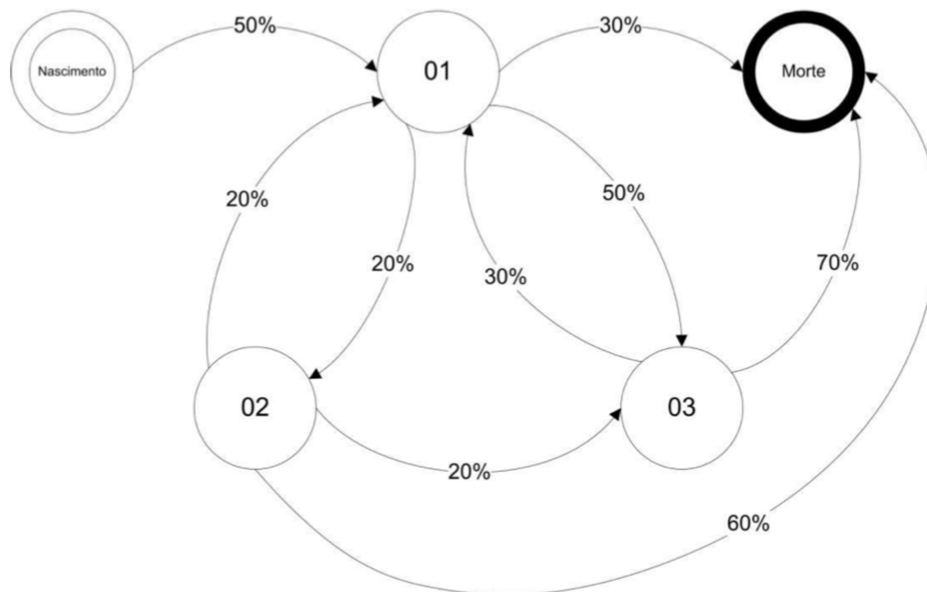

Figura 3.3: Grafo representativo do arquivo de configuração de modelo da Figura 3.2 (CRUZ, 2009)

A classe `NumeroAleatorios` é também uma classe abstrata e permite que se gere os números aleatórios de acordo com a distribuição definida pela subclasse concreta. O método abstrato responsável pela efetiva geração do número aleatório é o chamado `GerarProxNumero()`.

A classe abstrata `Comunicacao` é a responsável pela efetiva comunicação entre os processos envolvidos na simulação através do protocolo definido pela subclasse concreta, por exemplo, PVM ou MPI, conforme mencionado anteriormente. A classe `Mensagem` flexibiliza a implementação das possíveis mensagens que possam ser trocadas através da classe `Comunicacao`.

Cruz (2009) afirma que a classe `Protocolo` possui mecanismos que simplificam a

utilização dos protocolos de sincronização. Também é uma classe abstrata e a classe concreta correspondente é responsável por implementar o protocolo de sincronização, por exemplo, como já foi citado, o *Time Warp* ou o *Solidary Rollback*. A seguir serão descritos as principais operações da classe `Protocolo`:

- `IniciarEstruturas()`: responsável por preparar as estruturas do protocolo para sincronização.
- `InterromperSimulacao()`: permite que o usuário interrompa a simulação, podendo retomá-la posteriormente.
- `SalvarEstruturas()`: salva as estruturas do protocolo para, por exemplo, realizar uma migração.
- `GerarContabilidade()`: responsável por preparar uma lista de informações sobre o protocolo, como por exemplo, quantidade de *rollbacks*.
- `AtualizarEstruturas()`: permite atualizar as estruturas do protocolo com dados previamente configurados.

3.3 Modelagem dos Protocolos Otimistas *Solidary Rollback* e *Time Warp*

Cruz (2009) também propôs os diagramas de classes para as implementações dos protocolos *Solidary Rollback* (Figura 3.4) e *Time Warp* (Figura 3.5). Em ambos os diagramas, cada processo se inicia ao chamar o método `executar()` da classe `Processo`, que possui implementações nas classes concretas `ProcessoComObservador` e `Observador`.

O diagrama da Figura 3.4, foi inicialmente apresentado por Moreira (2005) como proposta de implementação para o *Solidary Rollback*. Tal implementação foi acoplada ao *framework*, por Cruz (2009). A classe `Checkpoint` é utilizada para marcar os *checkpoints* locais e pode ser redefinida utilizando o protocolo de marcação de *checkpoints* escolhido. No caso do diagrama em questão, é utilizado o FDAS, protocolo descrito no capítulo anterior desta dissertação. A implementação do processo observador, essencial para o funcionamento do *Solidary Rollback*, é dada pela classe `Observador_SR` que possui métodos para coordenar as linhas de recuperação, além de escalonar processos e realizar troca de protocolos.

O diagrama da Figura 3.5 foi baseado no da Figura 3.4, porém adaptado para o *Time Warp*, e também foi acoplado ao *framework*. Para o *Time Warp*, a classe `Checkpoint` é implementada através do mecanismo SSS (*Sparse State Saving*) (PREISS; MACINTYRE;

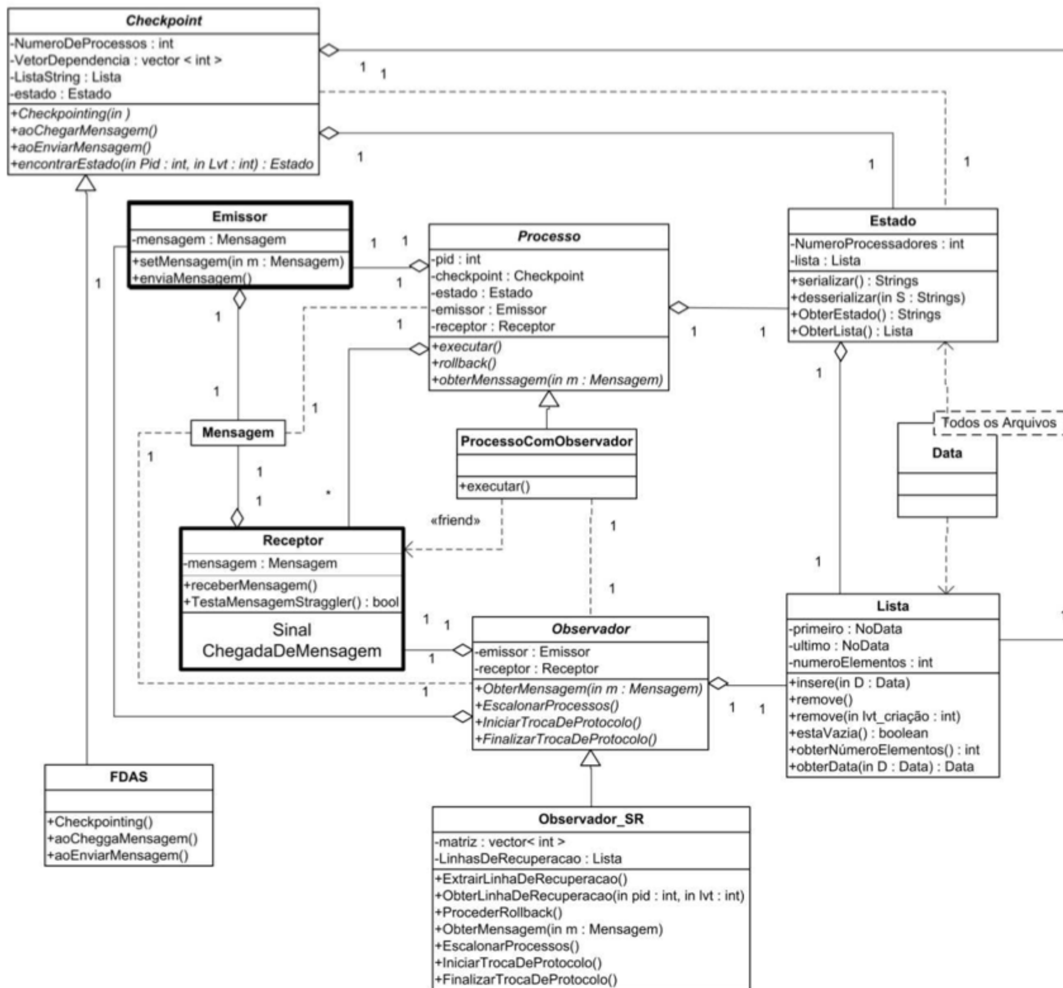

Figura 3.4: Diagrama de classes proposto para o *Solidary Rollback* (CRUZ, 2009)

LOUCKS, 1992). O processo observador não existe conceitualmente no *Time Warp*, porém, sua implementação através da classe `Observador_TW` tem a função de permitir a implementação de mecanismos para a troca dinâmica de protocolos ou a migração de processos.

3.4 Considerações Finais

Uma vez apresentado o *framework* proposto por Cruz (2009), o Capítulo 4 apresentará um *framework* alternativo, baseado no original, procurando possibilitar maior flexibilização de configuração e melhor encapsulamento das classes que tratam o protocolo.

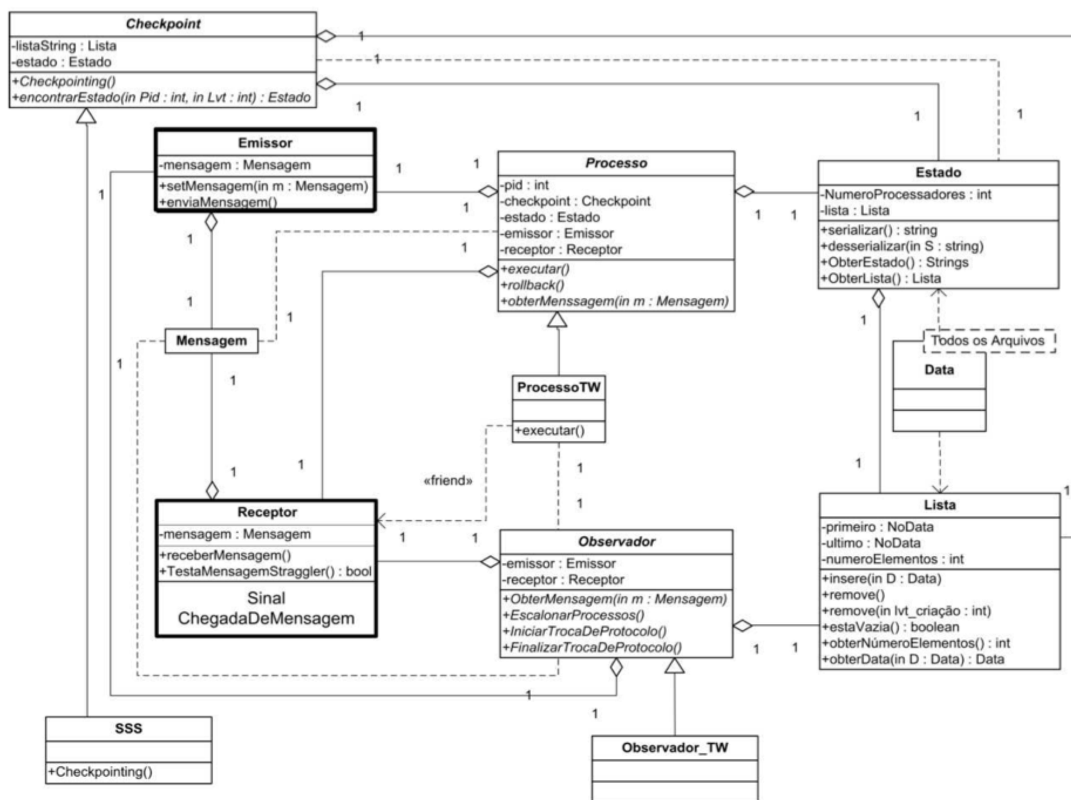

Figura 3.5: Diagrama de classes proposto para o *Time Warp* (CRUZ, 2009)

Capítulo 4

Adaptação e Descrição da Implementação do *Framework* de Simulação Distribuída

Este capítulo visa apresentar a descrição da implementação dos protocolos *Time Warp* e *Solidary Rollback*, com uma estrutura de classes similar à proposta por Cruz (2009), utilizando a implementação do protocolo MPI em Java[1] "MPJ Express" afim de aproveitar a flexibilidade inerente à tecnologia Java, além de permitir a implementação de simulação distribuída em um ambiente multiplataforma.

4.1 Escolha das tecnologias

De acordo com Geist et al. (1994), ao contrário do PVM, o MPI não tem a pretensão de ser uma infraestrutura de *software* completa e auto-suficiente que possa ser usada para computação distribuída. Logo, o MPI não inclui as funcionalidades de máquinas virtuais paralelas que podem ser encontradas no PVM, como gerenciamento de processos, configuração de máquina e suporte de entrada e saída. Assim, o MPI é melhor caracterizado como uma camada de interface de comunicação, que se enquadra facilmente no *framework* proposto por Cruz (2009), pois a arquitetura em camadas apresentada naquele trabalho prevê uma camada de comunicação. O MPI pode trazer em sua utilização algumas dificuldades quando for necessário realizar migração ou alocação dinâmica de processos. Junqueira (2012) discute em detalhe tais dificuldades.

Segundo Shafi e Manzoor (2009), o sistema de mensagens em Java "MPJ Express" é uma implementação de código aberto da API (Interface de Programação de Aplicativo, do

[1] Java é uma tecnologia criada pela Sun Microsystems, que foi posteriormente adquirida pela Oracle Corporation

inglês *Application Programming Interface*) mpiJava 1.2 (BAKER et al., 1999). A preparação do ambiente para teste em laboratório demonstrou que o MPJ Express se mostrou muito mais prático para instalar e configurar do que outras implementações MPI nativas de sistemas operacionais Linux, como o OpenMPI (GRAHAM et al., 2006), o LAM-MPI (BURNS; DAOUD; VAIGL, 1994) ou o MPI-CH (GROPP et al., 1996). Pelo fato de ser implementado em Java e independente da pré-existência de qualquer implementação nativa para sua execução, a não ser o código nativo (*shared libraries*) que já vem empacotado na sua instalação, ele pode ser utilizado em ambientes heterogêneos de plataforma, permitindo que cada processo envolvido na simulação possa executar em um sistema operacional diferente.

A escolha da tecnologia Java também permite a utilização de reflexão, que é o carregamento dinâmico de classes que inicialmente não estavam projetadas na solução, mas que foram agregadas a ela em tempo de execução. Assim, por exemplo, é possível parametrizar para o *framework* uma nova implementação de qualquer de suas classes abstratas e ela estará pronta para o uso, sem a necessidade de se alterar o código original do mesmo. Além disso, é possível utilizar tecnologias e linguagens conhecidas na literatura que a plataforma Java engloba, como arquivos XML (Linguagem de Marcação Extensível, do inglês *eXtensible Markup Language*).

4.2 Contribuição ao modelo para implementação em Java

Conforme será demonstrado, o *framework* aqui proposto possui classes comuns ao proposto por Cruz (2009), porém várias de suas partes foram refeitas, principalmente naquela que representa sua função principal, onde foi implementada a execução do modelo de simulação e o protocolo de sincronização. Desta forma, tem-se em mãos um novo *framework* que contribui para a comunidade científica como uma alternativa ao original proposto por Cruz (2009).

A Figura 4.1 apresenta a remodelagem do *framework* discutido no capítulo anterior para adaptar aos padrões conhecidos de codificação em Java, além de contribuir com uma reestruturação do tratamento dos protocolos de sincronização, separado da execução da simulação propriamente dita.

As classes apresentadas no diagrama da Figura 4.2 mantém a mesma estrutura apresentada em Cruz (2009). A seguir serão descritas as alterações mais importantes.

A Figura 4.3 apresenta a hierarquia de classes responsáveis por gerenciar os processos na simulação distribuída. A classe `Process` provê uma abstração para um processo alo-

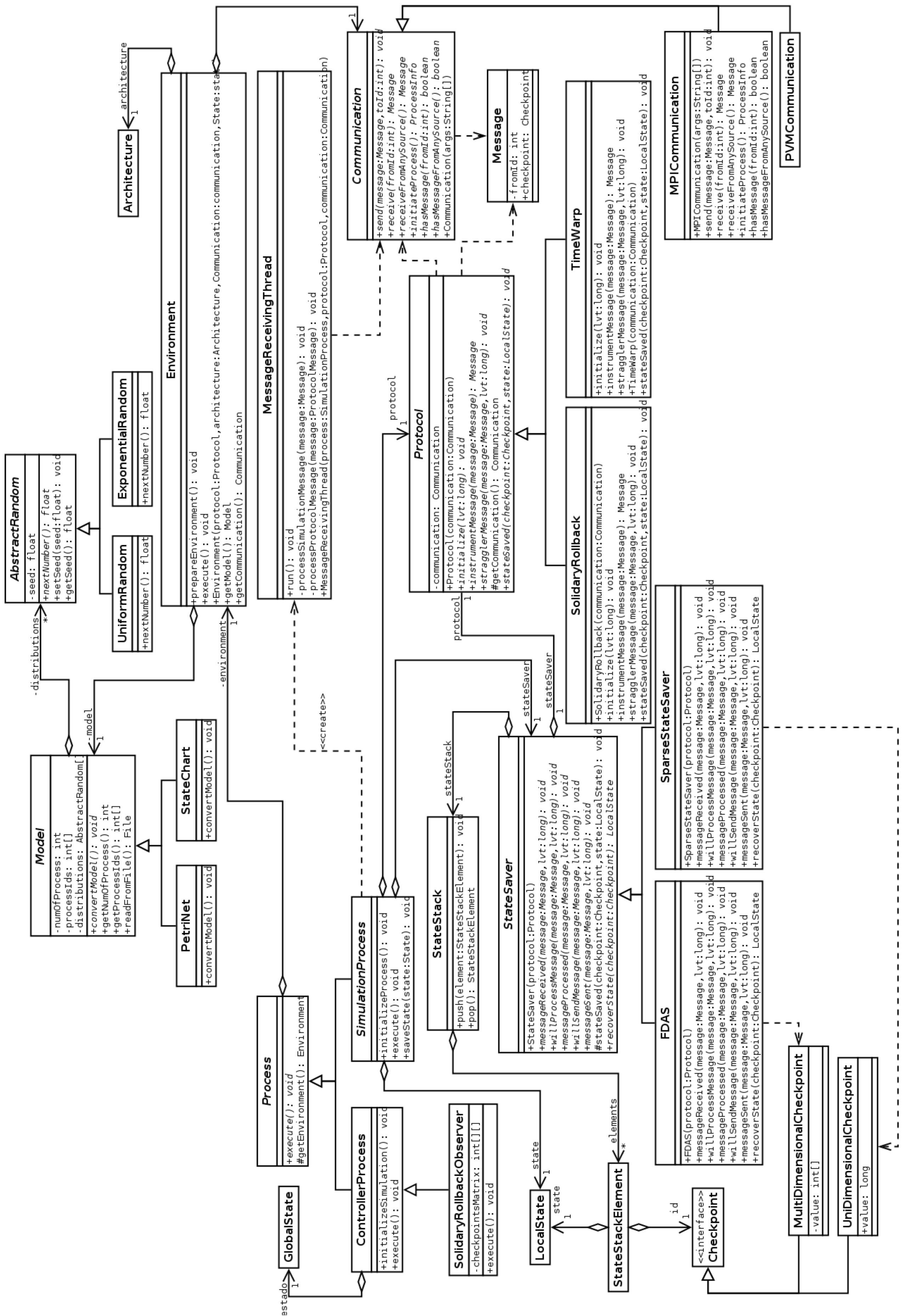

Figura 4.1: Diagrama de classes da remodelagem do *framework* proposto por Cruz (2009)

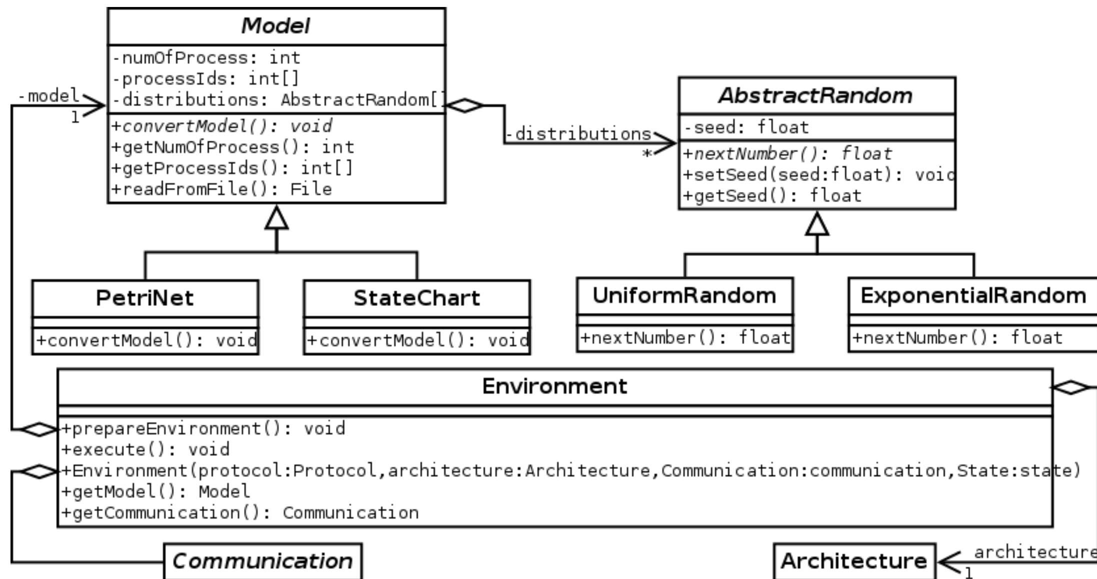

Figura 4.2: Classes inalteradas desde o *framework* proposto por Cruz (2009)

cado na execução da simulação, correspondente a um processo real executando uma tarefa, seja envolvida na simulação propriamente dita ou a algum controle. Provê, também, uma operação abstrata `execute()` onde a tarefa será propriamente implementada. Ainda contém a operação protegida abstrata `getEnvironment()` permitindo que as subclasses acessem o atributo privado `environment` e utilizem os serviços providos pela classe `Environment`.

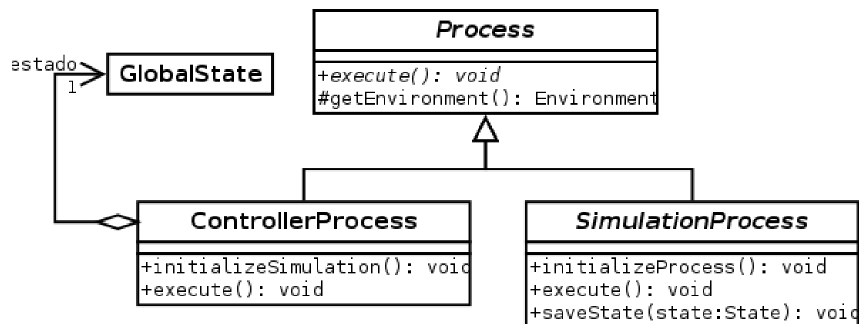

Figura 4.3: Classes responsáveis por gerenciar os processos envolvidos na simulação

A utilidade da classe `ControllerProcess` parte do pressuposto de que, seja qual for o protocolo de sincronização, deverá existir sempre um processo controlador que, no mínimo, irá coordenar o início da simulação, através da operação `initializeSimulation()`, centralizando a obtenção de dados do modelo e enviando os dados necessários para os processos efetivamente envolvidos na simulação. Para protocolos que demandem a existência de um processo observador, deve-se especializar a classe `ControllerProcess` para implementá-lo. A operação `execute()`, que originalmente não realiza nenhuma tarefa, na subclasse realizará as tarefas do observador. Não é uma operação abstrata na

superclasse, pois a mesma precisa ser concreta, permitindo execução sem especialização, para protocolos que não demandem observador.

A classe `SimulationProcess` é responsável por executar as tarefas de um processo da simulação. Todas as suas instâncias, nos diversos processos envolvidos na simulação, colaboram para que o modelo seja efetivamente executado. A operação `initializeProcess()` é responsável por receber as informações do processo controlador e preparar o ambiente para a simulação. A partir daí, a operação `execute()` é executada para realizar a simulação. Colaborando com a classe `SimulationProcess`, existe a classe `MessageReceivingThread`, instanciada pela primeira, responsável por monitorar os eventos que chegam naquele processo e redirecioná-los para o processo ou para a instância da classe que gerencia o protocolo de sincronização. As mensagens **de protocolo** ou **de processo** são identificadas por um valor inteiro vindo com a mensagem.

A classe `EstadoGeral` foi removida do modelo original para dar lugar a duas classes: `LocalState` e `GlobalState`. A primeira é responsável por manter as informações de estados locais de cada processo e a segunda é responsável por manter as informações de estado global da simulação. Como a classe `GlobalState` está associada à classe `ControllerProcess`, as instâncias da classe `SimulationProcess`, presentes em cada processo de simulação, deverão enviar, periodicamente, os dados de estado local de cada uma para que o controlador (`ControllerProcess`) possa compor o estado global da simulação. Sendo assim, as classes de estado não estão mais associadas à classe `Ambiente` (ou `Environment` na nova versão), mas sim a cada classe representativa de um processo, ou seja, `SimulationProcess` e `ControllerProcess`.

A Figura 4.4 apresenta as classes envolvidas no processo de salvamento de estados e a classe responsável pelo protocolo de sincronização. A classe `StateSaver` é responsável por realizar o salvamento de estados da simulação. É uma classe abstrata e suas subclasses deverão implementar o algoritmo de salvamento propriamente dito. A classe `SimulationProcess` é responsável por notificar a `StateSaver` sempre que eventos importantes provoquem o salvamento de estados. A recepção das notificações é representada pelas operações abstratas `messageReceived()`, `willProcessMessage()`, `messageProcessed()`, `willSendMessage()` e `messageSent()`, que correspondem, respectivamente, aos eventos de recepção de uma mensagem (antes da fila de eventos futuros), antes do processamento de uma mensagem (retirada da fila de eventos futuros), após o processamento de uma mensagem, antes de enviar uma mensagem para outro processo e após enviar uma mensagem para outro processo. Nem todas as notificações deverão representar um efetivo salvamento de estados, pois isso depende do algoritmo utilizado. Quando uma subclasse, implementando um determinado algoritmo, implementa o salvamento de estados para uma notificação específica, a operação `stateSaved()` da superclasse deverá ser chamada. A implementação desta última invocará a operação homônima da classe `Protocol`, que

será descrita a seguir.

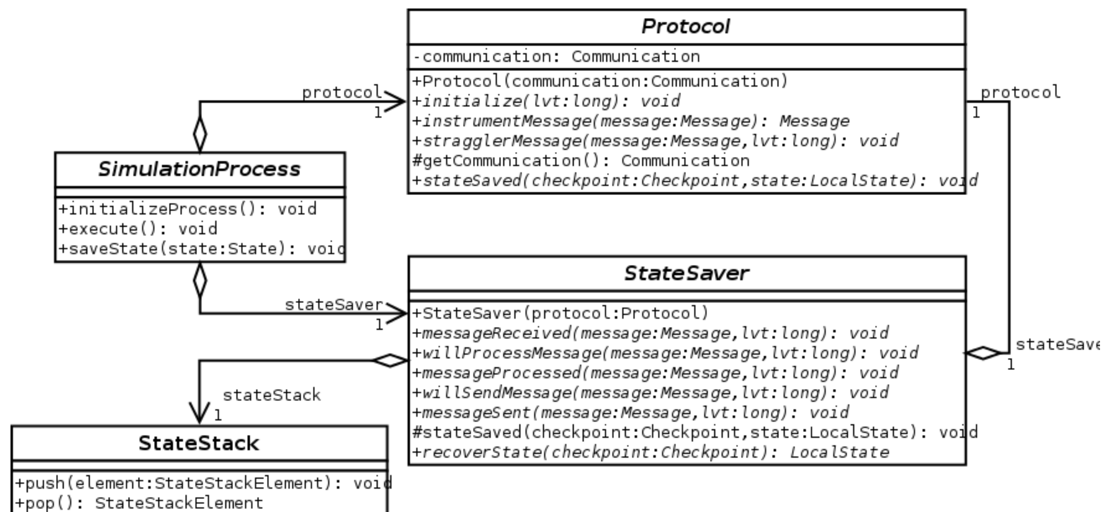

Figura 4.4: Classes envolvidas no salvamento de estados e responsáveis pelo protocolo de sincronização

O salvamento de estados é sempre realizado pela subclasse de `StateSaver` através da inclusão do estado local em uma pilha implementada pela classe `StateStack`. A classe `Protocol` também conhece a classe `StateSaver` para que, quando for necessário recuperar o estado correspondente a um *checkpoint* (seja um *checkpoint* multidimensional, no caso do *Solidary Rollback*[2], ou unidimensional, no caso do *Time Warp*[3]), a operação `recoverState` seja invocada para desempilhar o estado correspondente ao *checkpoint* da pilha `StateStack`.

A classe `Protocol` cuida de aspectos específicos dos protocolos de simulação. Ela recebe uma notificação de salvamento de estado da classe `StateSaver` ou então uma notificação de recebimento de uma mensagem *straggler* da classe `SimulationProcess`, situações representadas, respectivamente, pelas operações `stateSaved()` e `stragglerMessage()`. A operação `instrumentMessage()`, também será invocada pela instância de `SimulationProcess`, é responsável por "instrumentar" uma mensagem de simulação com os dados do protocolo, gerando uma mensagem de protocolo que será entendida pela instância de `Protocol` no processo receptor. Um exemplo de instrumentação é a inclusão do vetor de dependências associado a um *checkpoint* na mensagem.

A Figura 4.5 apresenta um diagrama de sequência ilustrando o cenário quando ocorre um novo evento, gerando uma notificação de salvamento de estados, e o cenário de chegada de uma mensagem *straggler*.

[2]O *checkpoint* utilizado no *Solidary Rollback* é aqui chamado de multidimensional pois é identificado pelo valor do relógio vetorial do processo no momento do salvamento de estado.

[3]O *checkpoint* utilizado no *Time Warp* é aqui chamado de unidimensional pois é identificado apenas pelo LVT do processo no momento do salvamento de estado.

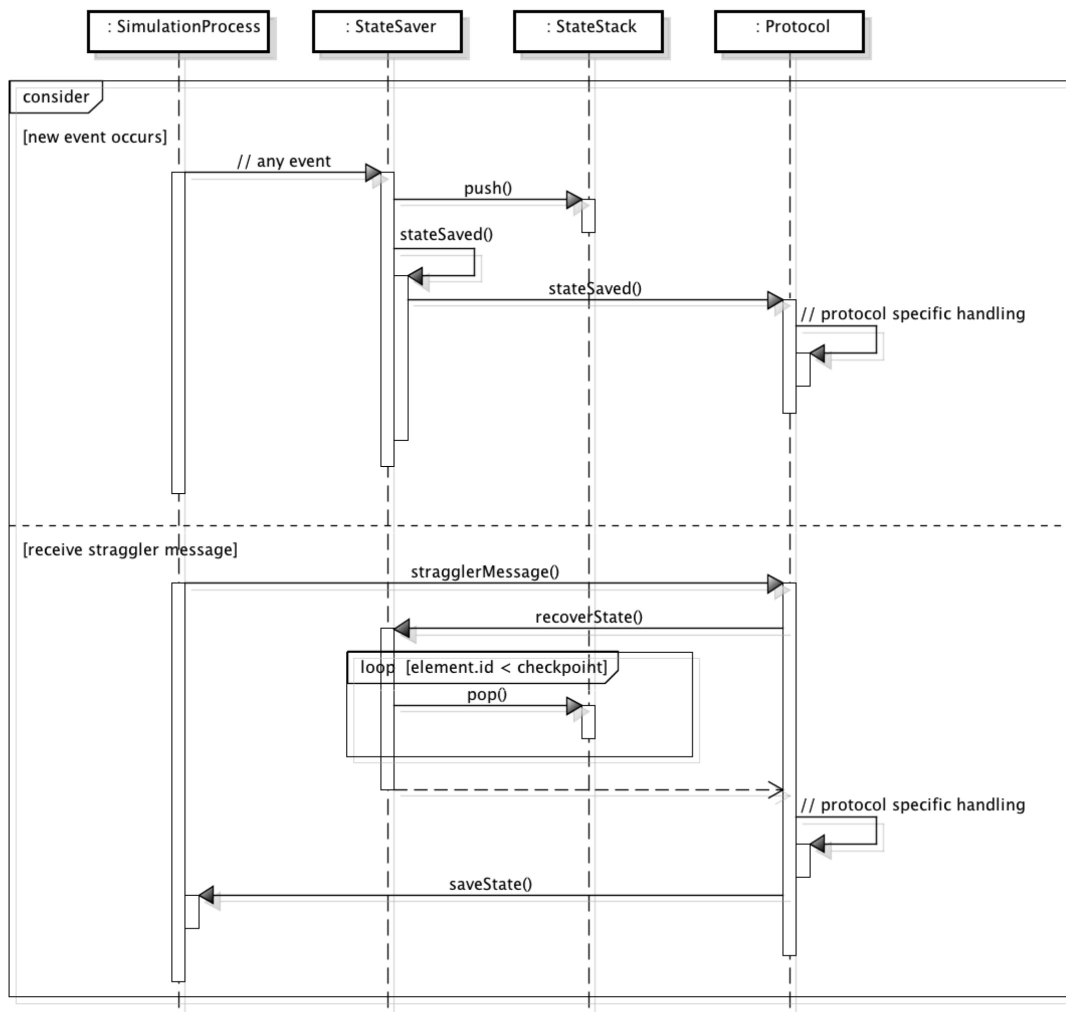

Figura 4.5: Salvamento e recuperação de estados

A primeira parte do diagrama de sequência se inicia quando ocorre uma situação que gere uma notificação da classe `SimulationProcess` para a classe especializada de `StateSaver`. Um exemplo de notificação é a chegada de uma mensagem na fila de eventos futuros ou o envio de uma mensagem para outro processo. Então, a especialização de `StateSaver` salva o estado na pilha implementada pela classe `StateStack`. Uma vez salvo, a classe especializada chama a operação `stateSaved()` da superclasse `StateSaver` que, por sua vez, invoca a operação homônima na classe `Protocol` para que os tratamentos relacionados ao protocolo sejam realizados. Um exemplo de tratamento é, no caso do *Solidary Rollback*, o envio das informações de *checkpoints* para o processo observador.

A segunda parte do diagrama ocorre quando uma mensagem *straggler* chega ao processo, levando a classe `SimulationProcess` a invocar a operação `stragglerMessage()` da classe `Protocol`. Esta última classe, por sua vez, invoca a operação `recoverState()` na classe especializada de `StateSaver` que recupera o estado da pilha `StateStack` e retorna para a classe `Protocol`. Tratamentos específicos do protocolo são realizados e o estado é

restaurado na classe `SimulationProcess` através da invocação da operação `saveState()`.

As classes `TimeWarp`, `SparseStateSaving` e `UniDimensionalCheckpoint` são responsáveis por implementar o protocolo *Time Warp*. As classes `SolidaryRollback`, `FDAS`, `SolidaryRollbackObserver` e `MultiDimensionalCheckpoint` são responsáveis pela implementação do protocolo *Solidary Rollback*. A classe `MPICommunication` corresponde à implementação do protocolo de troca de mensagens MPI e a classe `PVMCommunication` sugere a implementação da comunicação utilizando o PVM. O PVM não foi implementado no presente trabalho.

4.3 Aspectos práticos da implementação do *framework*

A tecnologia Java traz consigo uma gama de outras tecnologias que flexibilizam a implementação de uma solução. Exemplos podem ser a utilização de marcações em linguagem XML (*eXtensible Markup Language*), de arquivos de propriedades (*properties file*) e a possibilidade de identificar e instanciar objetos de classes em tempo de execução, tecnologia conhecida como *reflection* (do inglês "reflexão"), permitindo criar novas classes para o *framework* e apontá-las em arquivos de configuração.

Segundo Dashofy, Hoek e Taylor (2001), XML é uma linguagem de marcação extensível que se pretendia utilizar inicialmente para marcação de texto, porém seu uso crescente ocorre nas áreas de codificação de dados, serialização de objetos e gerência de metadados. Por ser extensível, novas *tags* podem ser criadas para incrementar a informação representada. A máquina virtual Java traz embarcada nativamente uma API para converter arquivos XML, não necessitando de bibliotecas adicionais, apesar de existirem muitas no mercado. O presente trabalho utiliza tal implementação nativa. A Listagem 4.1 apresenta um arquivo XML contendo as informações de um modelo a ser carregado no *framework*.

```
** Arquivo: model.xml **

<?xml version="1.0" encoding="UTF-8"?>
<model>
        <node>
                <id>1</id>
                <birthTax>50</birthTax>
                <nextNodeProbability>
                        <nextNode>2</nextNode>
                        <probability>80</probability>
                </nextNodeProbability>
                <dyingProbability>20</dyingProbability>
        </node>
```

```
    <node>
        <id>2</id>
        <birthTax>0</birthTax>
        <nextNodeProbability>
            <nextNode>3</nextNode>
            <probability>60</probability>
        </nextNodeProbability>
        <nextNodeProbability>
            <nextNode>4</nextNode>
            <probability>40</probability>
        </nextNodeProbability>
        <dyingProbability>0</dyingProbability>
    </node>
    <node>
        <id>3</id>
        <birthTax>0</birthTax>
        <nextNodeProbability>
            <nextNode>5</nextNode>
            <probability>100</probability>
        </nextNodeProbability>
        <dyingProbability>0</dyingProbability>
    </node>
    <node>
        <id>4</id>
        <birthTax>0</birthTax>
        <nextNodeProbability>
            <nextNode>5</nextNode>
            <probability>50</probability>
        </nextNodeProbability>
        <dyingProbability>50</dyingProbability>
    </node>
    <node>
        <id>5</id>
        <birthTax>0</birthTax>
        <dyingProbability>100</dyingProbability>
    </node>
</model>
```

Listagem 4.1: Modelo de simulação em XML

A documentação das *tags* do arquivo XML apresentado na Listagem 4.1 está descrita no Apêndice A.

A Figura 4.6 apresenta o modelo representado pelo arquivo XML sob a forma de diagrama de máquina de estados. Os estados do diagrama representam os processos e as transições representam a probabilidade de um atendimento passar de um processo para outro.

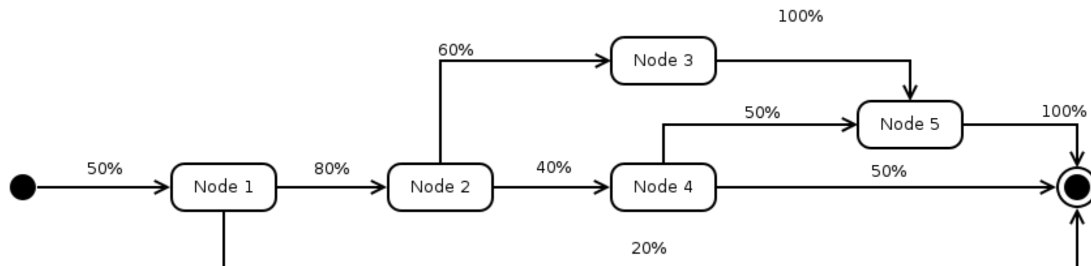

Figura 4.6: Diagrama de estados contendo o modelo representado pelo XML

O uso de arquivos de propriedade é amplamente difundido na tecnologia Java. Trata-se de um conjunto de informações sequenciais, cada qual contendo uma chave e um valor. Juntamente com a "reflexão", é possível configurar o *framework* com as classes que devem ser instanciadas tanto para o protocolo quanto para a comunicação. A Listagem 4.2 apresenta um arquivo de propriedades configurado para executar a simulação com o protocolo de sincronização *Solidary Rollback* e o protocolo de comunicação MPI. Assim, através da reflexão, as classes apropriadas serão instanciadas.

```
** Arquivo: configuration.properties **

# Synchronization protocol configuration
protocol.synchronization.class=edu.unifei.framework.SolidaryRollback
protocol.synchronization.controllerprocess.hasspecific=true
protocol.synchronization.controllerprocess.class=
edu.unifei.framework.SolidaryRollbackObserver

# State saving configuration
statesaving.class=com.unifei.framework.FDAS

# Communication protocol configuration
protocol.communication.class=edu.unifei.framework.MPICommunication
```

Listagem 4.2: Configuração dinâmica de protocolos

Segue a descrição das propriedades utilizadas:

- `protocol.synchronization.class`: indica a classe que implementa o protocolo de sincronização. No exemplo está a classe `edu.unifei.framework.SolidaryRollback` que implementa o protocolo *Solidary Rollback*.

- `protocol.synchronization.controllerprocess.hasspecific`: indica a existência de um processo controlador específico. Como o protocolo *Solidary Rollback* exige a existência de um processo observador, o valor da propriedade é configurado como `true` (verdadeiro).

- `protocol.synchronization.controllerprocess.class`: indica a classe específica para o processo controlador, sendo utilizada sempre que a propriedade anterior possuir valor `true`. No exemplo está a implementação do observador do *Solidary Rollback*, classe `edu.unifei.framework.SolidaryRollbackObserver`.

- `statesaving.class`: indica a classe de salvamento de estados. No exemplo está a classe `com.unifei.framework.FDAS`. Como o exemplo ilustra classes relacionadas com o *Solidary Rollback*, a classe de salvamento de estados deve utilizar *checkpoints* multidimensionais, pela utilização de *checkpoints* globais consistentes.

- `protocol.communication.class`: indica a classe que implementa o protocolo de comunicação entre os processos. No exemplo, o protocolo utilizado é o MPI, através da classe `edu.unifei.framework.MPICommunication`.

4.4 Considerações Finais

A utilização da tecnologia Java trouxe grande flexibilidade ao *framework*, principalmente através das técnicas de reflexão, permitindo a carga dinâmica de classes sem a necessidade de nova compilação. Tal flexibilidade motiva novas extensões que também tirem proveito da tecnologia Java.

O *framework* desenvolvido, apesar de baseado no proposto por Cruz (2009), possui diferenciais significativos, separando a execução do modelo de simulação do protocolo de sincronização e do algoritmo de salvamento de estados. Assim, é possível afirmar que o resultado é um novo *framework*.

Capítulo 5

Experimentos Realizados e Discussão de Resultados

O presente capítulo visa apresentar e discutir os resultados da aplicação do *framework* na simulação de modelos arbitrários elaborados pelo autor e modelos gerados aleatoriamente por um programa também elaborado pelo autor. Os modelos foram simulados de três formas diferentes: (1) através de um programa sequencial, (2) através do *framework* utilizando o protocolo de sincronização *Time Warp* e (3) através do *framework* utilizando o protocolo de sincronização *Solidary Rollback*. Para as simulações (2) e (3), o protocolo de comunicação utilizado foi o MPI. O objetivo das simulações foi comprovar que as simulações distribuídas (2) e (3) apresentam os mesmos resultados da simulação sequencial (1), além de comparar o *Time Warp* e o *Solidary Rollback* em termos do número de *rollbacks*, quantidade de eventos desfeitos (tamanho de cada *rollback*) e quantidade de estados salvos.

A Listagem 5.2 apresenta o pseudocódigo do programa sequencial de simulação, usado para validar as saídas dos protocolos. O programa lê o modelo de um arquivo XML, no formato especificado para o *framework* do Capítulo 4, e constrói a estrutura de processos. Para cada iteração, um evento é consumido da fila de eventos futuros, processado através do incremento do tempo virtual do processo e enviado para a fila do próximo processo. Ainda na iteração, um novo evento pode ser criado, de acordo com a taxa de natalidade, e inserido na fila de eventos futuros do próprio processo.

```
lerModeloDeArquivo(arquivoXML)

enquanto(verdadeiro) {
        paraCada(processo em listaProcessos) {
                se(filaEventos.vazia() = false) {
```

```
                        evento <- filaEventos.proximo()
                        tempoVirtual <- tempoVirtual + numeroAleatorio()
                        proximo <- processo.escolheProximo()
                        evento.setTempoVirtual(tempoVirtual)
                        proximo.inserirFilaEventos(evento)
                }

                evento <- gerarNovoEvento(natalidade)
                filaEventos.inserir(evento)
        }
}
```

Listagem 5.1: Pseudocódigo do programa sequencial de simulação

5.1 Modelos utilizados nas simulações

Afim de testar as implementações, foram utilizados modelos arbitrários e modelos gerados aleatoriamente. Os modelos aleatórios foram gerados através de um programa que se encontra descrito mais adiante neste capítulo.

5.1.1 Modelos arbitrários

Três modelos arbitrários foram utilizados para a comparação das implementações dos protocolos, a saber, utilizando 4 (quatro), 10 (dez) e 20 (vinte) processos. O primeiro modelo está representado pelo diagrama de estados da Figura 5.1.

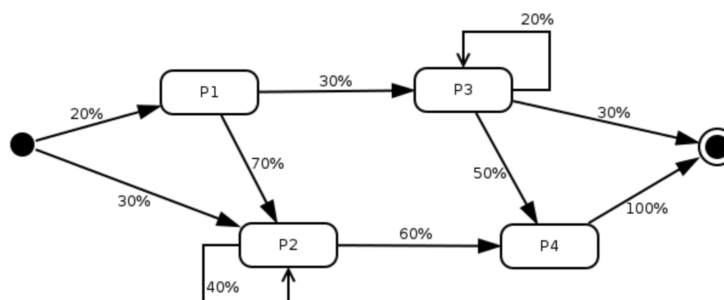

Figura 5.1: Modelo com 4 (quatro) processos

O segundo modelo utilizado equivale a um grafo bipartido, com 20% de probabilidade de envio de cada nó de um lado para cada nó do outro (vide Figura 5.2), sendo que, como ilustra a figura, nos nós da esquerda têm 20% de probabilidade de entrar um novo evento e todos os nós da direita enviam todos os eventos processados para a saída.

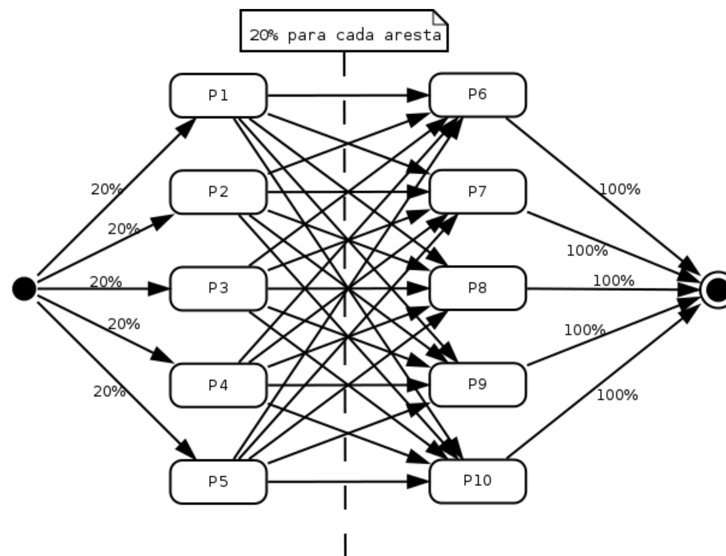

Figura 5.2: Modelo com 10 (dez) processos

O terceiro modelo, ilustrado pela Figura 5.3, é semelhante ao da Figura 5.2, porém utilizando quatro camadas (ou partições), ao invés de duas.

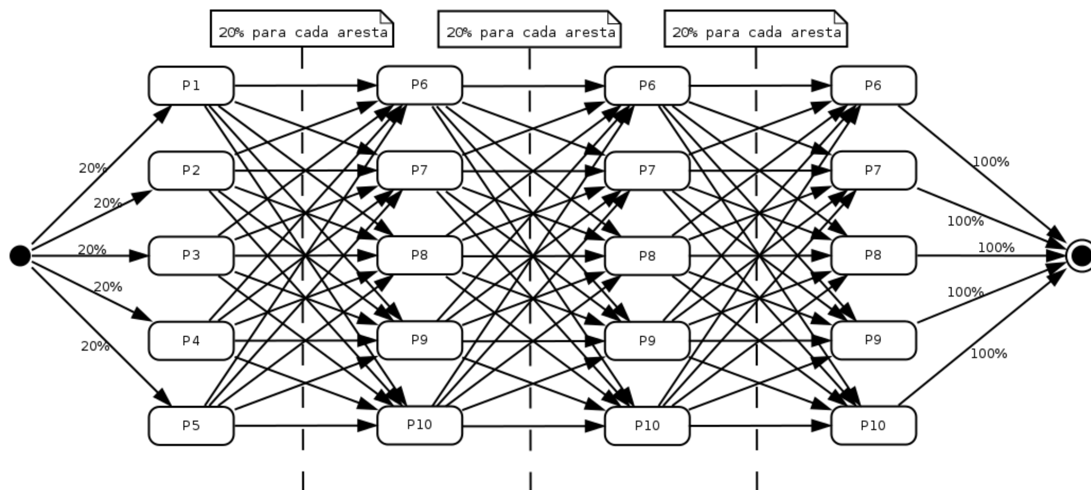

Figura 5.3: Modelo com 20 (vinte) processos

5.1.2 Modelos gerados aleatoriamente

Um programa de geração de modelos aleatórios foi utilizado para os demais modelos para comparar as implementações. O programa em questão utiliza os seguintes procedimentos para gerar o modelo:

Geração do modelo

1. O programa recebe como entrada o número total de processos, o número de processos

de entrada (com probabilidade de nascimento maior do que zero) e o número de processos de saída (com probabilidade de morte maior do que zero).

2. Dadas as entradas, o seguinte procedimento é executado:

   (a) Todos os processos são gerados e armazenados em um vetor.

   (b) Um subconjunto dos processos gerados acima é armazenado em um vetor de entradas, com tamanho igual ao número de entradas recebido como parâmetro. A probabilidade de nascimento é gerada aleatoriamente com um valor entre 1% e 100%.

   (c) Um subconjunto dos processos gerados acima é armazenado em um vetor de saídas, com tamanho igual ao número de saídas recebido como parâmetro. A probabilidade de morte é gerada aleatoriamente com um valor entre 50 e 100 (este valor será ponderado posteriormente). Existe a possibilidade de um mesmo processo pertencer ao vetor de entradas e ao de saídas.

3. Todos os processos são agrupados em subconjuntos, sendo que cada subconjunto contém pelo menos um processo de entrada e um de saída. O número de subconjuntos é dado pelo mínimo entre o número de processos de entrada e o número de processos de saída.

4. Para cada subconjunto:

   (a) Os processos do subconjunto que não pertençam ao vetor de entradas nem ao vetor de saídas são divididos em camadas. O número de camadas é dado pelo mínimo entre:

      - um número aleatório entre 1 e 4;
      - a metade do número de processos selecionados para serem divididos em camadas.

   (b) Caso o número de camadas resultante seja igual a zero e o número de processos selecionados seja maior do que zero, o número de camadas é então adotado como 1.

   (c) Cada processo de entrada do subconjunto é ligado a todos os processos da primeira camada.

   (d) Cada processo de uma camada anterior é ligado a todos os processos da próxima camada.

   (e) Cada processo da última camada é ligado a todos os processos de saída do subconjunto.

   (f) Caso não existam processos intermediários para o subconjunto, cada processo de entrada é ligado a todos os processos de saída.

   (g) A probabilidade de cada ligação é dada por um valor aleatório entre 50 e 100 (este valor será ponderado posteriormente).

5. O processo acima assegura que para qualquer entrada existe sempre um caminho para a saída e existe sempre um caminho entre uma entrada e uma saída passando por qualquer processo do modelo.

6. Novas ligações aleatórias são geradas entre quaisquer pares de processos do modelo, independente do subconjunto ou das camadas aos quais pertençam. O número de novas ligações geradas é dado por um valor aleatório entre 1 e o número total de processos do modelo.

7. A probabilidade de cada nova ligação é dada por um valor aleatório entre 50 e 100 (este valor será ponderado posteriormente).

Ponderação do modelo

1. Para cada processo do modelo:

   (a) É somada a probabilidade de saída para todos os processos aos quais o processo se encontre ligado. A este valor é somada a probabilidade de morte do processo.

   (b) Cada probabilidade é recalculada seguindo a fórmula:

$$P_{nova} = \frac{P_{anterior} * 100}{P_{total}} \tag{5.1}$$

   Onde:

   - P_{nova} é a nova probabilidade da ligação ou de morte do processo
   - $P_{anterior}$ é a probabilidade anterior de ligação ou de morte do processo, antes de ser ponderada
   - P_{total} é a probabilidade total, somando todas as probabilidades de saída do processo e a probabilidade de morte

2. Ao final da ponderação, a soma de todas as probabilidades de saída e da probabilidade de morte de cada processo deve ser igual a 100.

Validação e geração do arquivo de saída

1. O modelo é validado para verificar se as probabilidades de saída e de morte somam 100%. Caso não seja, um erro é gerado e o programa é encerrado.

2. O arquivo de saída é gerado no formato XML conforme especificado para o *framework* do Capítulo 4, podendo ser utilizado como entrada tanto para o *framework* quanto para o programa sequencial.

Para exemplificar, a Figura 5.4 apresenta um diagrama de estados representando um modelo gerado pelo programa gerador de modelo. Para o exemplo, foram considerados 10 processos no total, 2 processos de entrada e 3 processos de saída. Nota-se que a soma da probabilidade de saída de cada processo corresponde a 100%. Para qualquer entrada existe sempre um caminho para a saída. As ligações aleatórias do passo 6 do item **Geração do modelo** asseguram que o modelo final não possui a característica de se apresentar em camadas (veja passo 4, item (b) do item **Geração do modelo**).

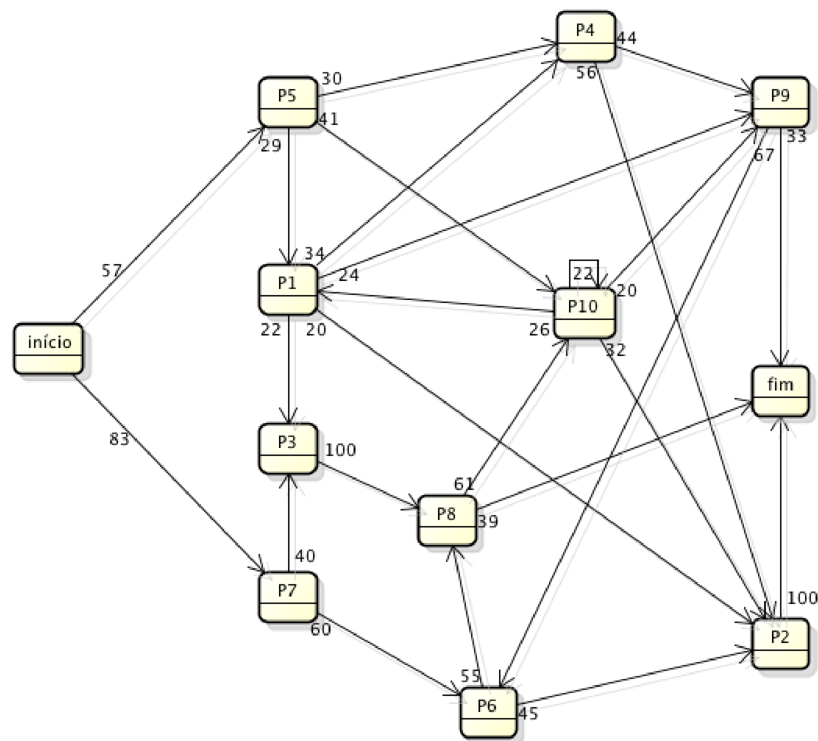

Figura 5.4: Exemplo de modelo gerado

Três modelos distintos foram gerados para testar as implementações, todos com número total de processos igual a 50. O primeiro possui 10 processos de entrada e 20 processos de saída, o segundo 20 processos de entrada e 10 processos de saída e o terceiro possui 20 processos de entrada e 20 processos de saída.

5.2 Resultado das simulações

As execuções do *Time Warp* e do *Solidary Rollback*, utilizando o *framework* proposto no Capítulo 4, apresentaram o mesmo resultado em termos de saída do modelo (ordenação

de eventos e tempo virtual, ambos na saída) que o programa de simulação sequencial, validando as implementações. A Listagem 5.2 apresenta um trecho de um arquivo de saída de exemplo. Trata-se de um arquivo CSV (valores separados por vírgula) em que cada linha representa um evento saindo da aplicação sequencial (ou do framework) e os valores de cada linha correspondem, respectivamente, ao número do processo onde o evento iniciou, o número sequencial do evento gerado no processo inicial e o valor do LVT no processo de saída.

```
...
17,92,17862
2,181,18318
2,206,20317
34,196,20389
22,163,20452
22,183,22694
34,226,23612
34,231,24289
34,242,24951
34,244,25125
22,199,25203
34,247,25600
22,208,26214
9,85,26574
17,132,27025
34,265,27407
...
```

Listagem 5.2: Pseudocódigo do programa sequencial de simulação

A Tabela 5.1 apresenta os dados de eficiência dos protocolos para cada modelo. Tais dados são o número de *rollbacks*, a quantidade de eventos desfeitos e a quantidade de eventos salvos. Os dados foram totalizados a cada 1000 (mil) LVTs. As colunas **M1**, **M2** e **M3** se referem, respectivamente, aos modelos de quatro, dez e vinte processos. As colunas **G1**, **G2** e **G3** se referem aos três modelos gerados pelo programa gerador de modelos aleatórios.

A implementação dos mecanismos de tomada de *checkpoints* para ambos os protocolos se realiza a cada 100 unidades de tempo virtual (LVT). Para o *Solidary Rollback* outros *checkpoints* são tomados através do mecanismo FDAS.

Após das medidas comparativas, considerando-se o número de eventos salvos e o número de eventos descartados, pode-se perceber que o protocolo *Time Warp* mostrou-se mais eficientes executando modelos mais simples, como os modelos **M1** e **M2**. Para

Protocolo	Medida	M1	M2	M3	G1	G2	G3
Dados comuns aos dois protocolos	Número de processos	4	10	20	50	50	50
	Número de entradas	2	5	5	10	20	20
	Número de saídas	2	5	5	20	10	20
Solidary Rollback	Número de *rollbacks*	90	29	316	125	83	67
	Eventos salvos	215	962	973	2019	3040	3311
	Eventos descartados	789	190	866	1076	1666	2278
	Salvos / Descartados	0,27	5,06	**1,12**	**1,88**	**1,82**	**1,45**
Time Warp	Número de *rollbacks*	333	72	2162	2075	5057	2853
	Eventos salvos	242	967	967	1383	2647	2350
	Eventos descartados	378	139	1527	1670	4397	2920
	Salvos / Descartados	**0.64**	**6.96**	0.63	0.83	0.60	0.80

Tabela 5.1: Comparativo dos protocolos *Time Warp* e *Solidary Rollback*

modelos mais complexos, como os modelos **M3**, **G1**, **G2** e **G3**, o protocolo *Solidary Rollback* se mostrou mais eficiente.

Em termos de número de *rollbacks*, o protocolo *Solidary Rollback* se mostrou mais eficiente para todos os modelos, pois ocasionou significativamente menos *rollbacks* do que o protocolo *Time Warp*.

5.3 Considerações finais

Conclui-se então que ambos os protocolos tiveram o mesmo resultado da simulação sequencial no que diz respeito ao resultado final da simulação. Quanto ao número de eventos salvos por eventos desfeitos, o *Solidary Rollback* se mostrou mais eficiente que o *Time Warp* para modelos mais complexos, enquanto o *Time Warp* foi mais eficiente para modelos mais simples. Quanto ao número de *rollbacks*, o *Solidary Rollback* foi, em geral, mais eficiente.

Capítulo 6

Conclusão

Foi apresentada, neste trabalho, a construção de um *framework* para simulação distribuída, em linguagem Java, utilizando a implementação MPJ-Express do protocolo de comunicação MPI. A implementação teve como objetivo demonstrar e comparar o funcionamento dos protocolos otimistas de sincronização de simulação distribuída *Time Warp* e *Solidary Rollback*.

6.1 Contribuições

O *framework* implementado baseou-se na proposta de Cruz (2009), porém diversas alterações foram adicionadas, além da implementação em linguagem Java, pois a proposta inicial foi baseada na linguagem C++. A principal contribuição foi a separação da execução do modelo pelo processo participante da simulação do protocolo de sincronização, permitindo maior flexibilidade ao *framework*. Devido a tal separação, é possível escolher dinamicamente o protocolo de sincronização utilizado em tempo de execução através de reflexão, característica da tecnologia Java que permite que classes sejam carregadas em tempo de execução. Assim, novos protocolos poderão ser implementados e sua instalação consistirá apenas na inclusão da biblioteca em local apropriado, alteção do arquivo de configuração (arquivo de propriedades) e reinício da aplicação baseada no *framework*, sem a necessidade de compilar todo o código novamente.

Outra contribuição foi a alteração dos formatos dos arquivos de configuração do modelo e da arquitetura para a linguagem XML, permitindo possíveis extensões futuras, pela própria característica extensível da linguagem.

6.2 Análise comparativa

Para a obtenção dos resultados, foram criados modelos hipotéticos e utilizados modelos gerados por um gerador de modelos aleatórios. Eles foram simulados em um programa sequencial, executados no *framework* configurado para o *Time Warp* e no *framework* configurado para o *Solidary Rollback*. Pode-se demonstrar que ambas as execuções do *framework* apresentaram o mesmo resultado que o programa sequencial. O *Solidary Rollback* foi mais eficiente do que o *Time Warp*, em termos de eventos salvos por descartados, para os modelos mais complexos, enquanto o *Time Warp* foi mais eficiente nos mais simples. Em termos de número de *rollbacks*, o *Solidary Rollback* gerou significativamente menos *rollbacks* do que o *Time Warp* para todos os modelos simulados.

6.3 Trabalhos futuros

Como trabalhos futuros, poderá ser implementada a troca dinâmica de protocolos, a integração da implementação em Java com a aplicação de modelagem apresentada por Barbosa (2012), a implementação do protocolo de comunicação baseado em PVM e a migração de processos semelhante ao realizado em Junqueira (2012), onde poderá ser agregado recursos que a linguagem Java apresenta como a possibilidade de transmitir o *bytecode* (código Java compilado) de um processo para outro.

A implementação em linguagem Java do *framework*, bem como as contribuições que o presente trabalho apresenta, devidamente validadas aqui, colaboram com a construção de ferramentas de simulação distribuída de eventos discretos, em especial com a utilização de protocolos otimistas de sincronização. Tomando como base o presente trabalho, novas pesquisas poderão se realizadas tirando proveito do *framework*, cujo código será disponibilidazado para a comunidade científica no acervo da Unifei.

Referências Bibliográficas

BAKER, M. et al. mpijava: An object-oriented java interface to mpi. *the 1st Java Workshop at the 13 th IPPS and 10th SPDP Conference*, April 1999.

BANKS, J. Introduction to simulation. *Proceedings of The 1999 Winter Simulation Conference*, p. 7–13, 1999.

BARBOSA, J. P. C. *Uma Ferramenta Paralela para Simulação de Eventos Discretos com Monitoramento Dinâmico de Processos*. 2012.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. *The Unified Modeling Language User Guide*. 2nd. ed. [S.l.]: Addison-Wesley Professional, 2005.

BRYANT, R. E. *Simulation of Packet Communication Architecture Computer Systems*. [S.l.]: Massachusetts Institute of Technology, 1977.

BUI, P. T.; LANG, S.-D.; WORKMAN, D. A. A conservative synchronization protocol for dynamic wargame simulation. *2003 Spring Simulation Interoperability Workshop*, 2003.

BURNS, G.; DAOUD, R.; VAIGL, J. Lam: An open cluster environment for mpi. *Proceedings of Supercomputing Symposium*, p. 379–386, 1994.

CHANDY, K. M.; MISRA, J. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, p. 440–452, 1978.

CHWIF, L.; MEDINA, A. C. *Modelagem e Simulação de Eventos Discretos – Teoria e Aplicações*. Segunda edição. [S.l.]: Editora Bravarte, 2007.

CRUZ, L. B. da. *Projeto de Um Framework para o Desenvolvimento de Aplicações de Simulação Distribuída*. 2009.

DASHOFY, E. M.; HOEK, A. van der; TAYLOR, R. N. A highly-extensible, xml-based architecture description language. *Conference on Software Architecture, 2001. Proceedings. Working IEEE/IFIP*, p. 103–112, 2001.

FUJIMOTO, R. M. Parallel and distributed simulation. *Proceedings of the 1999 Winter Simulation Conference*, p. 122–131, 1999.

FUJIMOTO, R. M. *Parallel and Distributed Simulation Systems*. [S.l.]: Wiley–Interscience, 2000.

GEIST, A. et al. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing.* [S.l.]: The MIT Press, 1994.

GRAHAM, R. L. et al. Open mpi: A high-performance, heterogeneous mpi. *IEEE International Conference on Cluster Computing*, p. 1–9, September 2006.

GROPP, W. et al. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Computing*, v. 22, p. 789–828, 1996.

JEFFERSON, D. R. Virtual time. *ACM Transactions on Programming Languages and Systems*, v. 7, n. 3, p. 404–425, July 1985.

JUNQUEIRA, M. A. F. C. *Mecanismos para Migração de Processos na Simulação Distribuída.* 2012.

LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, v. 7, n. 21, p. 558–565, July 1978.

MALIK, A. W.; PARK, A. J.; FUJIMOTO, R. M. Optimistic synchronization of parallel simulations in cloud computing environments. *IEEE International Conference on Cloud Computing*, p. 49–56, September 2009.

MANIVANNAN, D.; SINGHAL, M. Quasi-synchronous checkpointing: Models, characterization, and classification. *IEEE Transactions on Parallel and Distributed Systems*, v. 10, n. 7, p. 703–713, July 1999.

MELLO, B. A. de. *Modelagem e Simulação de Sistemas.* Universidade Regional Integrada do Alto Uruguai e das Missões, 2001. Disponível em: <http://www.munif.com.br/munif2/arquivos/ap-sim.pdf?id=319>.

MOREIRA, E. M. *Rollback Solidário: Um Novo Protocolo Otimista para Simulação Distribuída.* Tese (Doutorado) — USP – São Carlos, SP, 2005.

MOREIRA, E. M.; SANTANA, R. H. C.; SANTANA, M. J. Using consistent global checkpoints to synchronize processes in distributed simulation. *Proceedings of the 2005 Ninth IEEE International Symposium on Distributed Simulation and Real–Time Applications*, p. 43–50, 2005.

NANCE, R. E.; SARGENT, R. G. Perspectives on the evolution of simulation. *Electrical Engineering and Computer Science*, n. Paper 100, 2002. Disponível em: <http://surface.syr.edu/eecs/100>.

NETZER, R. H. B.; XU, J. Necessary and sufficient conditions for consistent global snapshots. *IEEE Transactions on Parallel and Distributed Systems*, v. 6, n. 2, p. 165–169, February 1995.

PREISS, B. R.; MACINTYRE, I. D.; LOUCKS, W. M. On the trade-off between time and space in optimistic parallel discrete-event simulation. *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*, p. 33–42, January 1992.

SAMADI, B. *Distributed Simulation.* Tese (Doutorado) — University of California, Los Angeles, 1985.

SHAFI, A.; MANZOOR, J. Towards efficient shared memory communications in mpj express. *IEEE International Symposium on Parallel and Distributed Processing*, May 2009.

WALKER, D. W. An mpi version of the blacs. *In Proceedings of the 1994 Scalable Parallel Libraries Conference*, October 1994.

WANG, Y.-M. Consistent global checkpoints that contain a given set of local checkpoints. *IEEE Transactions on Computers*, v. 46, n. 4, p. 456–468, April 1997.

Apêndice A

Descrição das *tags* do XML representativo de um modelo

- **model:** A *tag* `model` é a *tag* principal do XML de modelo, sendo única por arquivo e contendo toda a estrutura de um modelo.

- **node:** A *tag* `node` deve ser posicionada abaixo da *tag* `model` e representa um processo na simulação. Ela possui as *subtags* `id`, representando a identificação do processo, `birthTax`, identificando a taxa de natalidade de novos eventos no processo, `nextNodeProbability`, identificando a probabilidade de enviar mensagens para outros processos e `dyingProbability`, identificando a taxa de eventos que deixam a simulação após o tratamento pelo processo identificado pela *tag* `node`.

- **nextNodeProbability:** A *tag* `nextNodeProbability`, conforme foi dito anteriormente, identifica a probabilidade de um processo enviar uma mensagem para outro processo. Pode se repetir dentro da *tag* `node` representando a probabilidade de envio para mais de um processo. Para cada `node`, só pode haver uma *tag* `nextNodeProbability` para cada processo da simulação. Ela possui as *subtags* `nextNode`, contendo a identificação do próximo processo e `probability`, informando a probabilidade de envio para o processo identificado pelo `nextNode`. Caso o valor de `nextNode` seja igual ao valor da *tag* `id` do `node`, indica uma probabilidade de envio de mensagem para o próprio processo remetente.