Universidade Federal de Itajubá
Programa de Pós-Graduação em Engenharia Elétrica

Projeto de Um *Framework* para o Desenvolvimento de Aplicações de Simulação Distribuída

Liverson Batista da Cruz

Orientador Edmilson Marmo Moreira

Co-orientador Otávio Augusto Salgado Carpinteiro

UNIFEI – Itajubá

Julho de 2009

Projeto de Um *Framework* para o Desenvolvimento de Aplicações de Simulação Distribuída

Liverson Batista da Cruz

Dissertação submetida ao Programa de Pós-Graduação em Engenharia Elétrica como parte dos requisitos para obtenção do Título de **Mestre em Ciências em Engenharia Elétrica.**

Orientador Edmilson Marmo Moreira

Co-orientador Otávio Augusto Salgado Carpinteiro

UNIFEI – Itajubá

Julho de 2009

Dedico este trabalho

aos meus pais Geraldo e Laura,

à minha esposa Ana Paula.

Agradecimentos

Agradeço primeiramente a Deus, que me abençoou em todos os momentos da minha vida pessoal e profissional.

Ao meu orientador Edmilson Marmo Moreira pela confiança, paciência e apoio. Também pela disponibilidade em vários feriados e finais de semana em que deixou sua família para trabalharmos na conclusão deste trabalho.

Ao meu orientador Otávio Augusto Salgado Carpinteiro pela credibilidade e oportunidade de realizar este trabalho.

Ao professor Enzo Seraphim pela amizade e pelo auxílio em outras atividades que enriqueceram meu conhecimento técnico, científico e profissional.

Aos meus pais Geraldo e Laura pelo incentivo e dedicação em todos os momentos da minha vida.

À minha esposa Ana Paula, pela compreensão e apoio durante todos estes anos em que a distância fez parte do nosso relacionamento.

Aos meus irmãos: Luciano, Lino, Lauro, Lisie e Letícia pelo carinho, pela torcida e pela amizade tão presente em nossa família.

Aos companheiros da república “Manicômyou” pela amizade e pelo convívio de anos dividindo o mesmo espaço.

Aos meus amigos Denis de Carvalho Braga, Renato Takahashi, Marco Antonio C. Craveiro, Rafael V. Molina, Vinicius C. Scarpa e Tamatiá R. Colmán Aveiro pela amizade, companheirismo e trocas de experiência.

Aos meus amigos Thiago de Almeida Crespo, Luiz Carlos Rodrigues e Guilherme Ferreira da Mata pela participação fundamental nos resultados de trabalho.

Aos professores do GPESC por permitir em que eu utilizasse inúmeras vezes os laboratórios para a realização dos experimentos.

Sumário

Lista de Figura

Resumo

A utilização de *frameworks*, para auxiliar o desenvolvimento de ferramentas de *software*, é uma realidade no mundo de hoje. Visualizando o cenário atual, este trabalho apresenta um *framework* que oferece suporte ao desenvolvimento de programas de simulação distribuída. O diferencial desta ferramenta, em relação às outras que auxiliam a programação paralela, é a quantidade de recursos disponíveis. Parte deles está relacionada à possibilidade de utilizar diferentes tipos de protocolos de sincronização em uma mesma aplicação e utilizar bibliotecas de comunicação diferentes, como o *PVM* (*Parallel Virtual Machine*) e o *MPI* (*Message Passing Interface*).

Para o projeto das classes do *framework* foram considerados dois protocolos de sincronização otimistas: *Time Warp* e *Rollback* Solidário. Apesar de pertencerem à mesma classe, estes protocolos possuem comportamentos diferentes. Assim, uma discussão comparativa é apresentada.

Projetado, utilizando programação orientada a objeto, este *framework* possui classes flexíveis e reutilizáveis que permitem futuras adaptações e extensão.

Capítulo 1: Introdução

Por várias décadas simulações são utilizadas para identificar comportamentos de sistemas complexos que envolvem diversas áreas científicas como: a Computação, a Física, a Matemática e a Química. Vários cientistas, através dos anos, utilizaram os resultados obtidos em simulações para tomar decisões referentes a projetos e pesquisas. Hoje em dia, até homens de negócios utilizam estes dados para prever orçamentos e realizar investimentos.

Uma simulação é caracterizada pela geração de um histórico artificial do comportamento de algum sistema. A observação deste histórico permite ao pesquisador ou analista criar inferências concernentes às características operacionais do sistema real simulado (BANKS,1999). A geração destes históricos é realizada através de modelos de sistemas criados especificamente para descrever um sistema particular. O modelo de um sistema é uma representação simplificada de um conjunto estruturado de componentes que interagem de modo regular entre si e com o meio ambiente (PERIN, 1995), sendo classificado de várias maneiras de acordo com suas características. Desta forma, um modelo pode ser: físico ou matemático, fechado ou aberto, analítico ou numérico, estático ou dinâmico, determinístico ou estatístico, contínuo ou discreto. Atualmente, todos os sistemas podem ser modelados parcialmente ou totalmente, através de equações matemáticas, e a utilização

destes modelos em simulações são úteis e permitem a obtenção de dados reais com menor custo, sem a utilização de protótipos e com uma margem de segurança aceitável. Simulações permitem também prever situações adversas com um nível alto de percepção, e isto ocorre devido à quantidade de informações obtidas durante e ao final do experimento.

Hoje em dia as simulações são realizadas principalmente em sistemas computacionais visando agilidade na obtenção dos dados. Estes sistemas podem possuir duas configurações: centralizada e distribuída.

1.1. Sistema Centralizado e Sistema Distribuído

Há pouco mais de uma década, os sistemas computacionais eram, em sua maioria, sistemas centralizados. A maioria das organizações possuía poucos computadores e, por falta de meios de interconexão, estes operavam separadamente. Vários avanços tecnológicos, envolvendo processadores e formas de interconexão de computadores, começaram a mudar este panorama, destacando-se o desenvolvimento de poderosos microprocessadores, alguns com poder computacional comparável a *mainframes*, a custo mais acessível; e o aumento da capacidade computacional desses, que a cada 18 meses vêm dobrando seu desempenho na última década; e continuam a evoluir em uma taxa muito maior que supercomputadores.

A tecnologia de interconexão de computadores evoluiu de forma similar. Difundiu-se o uso de redes de computadores, à medida que estas se tornavam mais confiáveis e permitiam maiores taxas de transmissão. Esta combinação de fatores levou ao desenvolvimento de uma nova forma de organização dos sistemas empregando redes de computadores. Estes são geralmente chamados sistemas distribuídos, em contraste aos sistemas centralizados anteriormente citados.

Não existe uma única definição para Sistemas Distribuídos *(SD)*. Uma definição precisa e completa do que é um sistema distribuído é difícil de

apresentar, visto que cada autor na literatura disponível apresenta definições próprias e com diversos pontos conflitantes.

Lamport (1982) define *SD* como aquele que impede a obtenção de quaisquer serviços, quando há falha em alguma máquina do sistema que o usuário nem imaginava que existia. Já Tanenbaum (1995), para definir o que é um *SD*, apresenta o conceito de transparência, em que um conjunto de computadores é utilizado como se fosse apenas um único computador. Enquanto, Enslow (1978) considera que um sistema só pode ser caracterizado como um *SD* se ele possuir simultaneamente cinco ingredientes básicos, sendo eles: multiplicidade de recursos de uso geral, distribuição física destes recursos com interação entre eles, existência de um sistema operacional de alto nível, que unifique e integre o controle dos componentes distribuídos, transparência do sistema e autonomia cooperativa de todos os recursos. Além destas definições alguns “sintomas” que podem caracterizar um *SD* são apresentados por Schroeder (1993), tais como: a existência de múltiplos elementos de processamento, *hardware* de interconexão e estados compartilhados, além do fato de que falhas dos elementos de processamento podem ocorrer de forma independente. Analisando as definições e “sintomas” apresentados pode-se perceber que poucos sistemas podem ser considerados como sistemas distribuídos, porém, a definição proposta por Tanenbaum (1995) adicionada a características de tolerância a falhas é a mais abrangente.

1.2. Simulação Distribuída

A Simulação Distribuída consiste em utilizar um Sistema Distribuído para simular modelos complexos visando um melhor desempenho na obtenção dos resultados de uma simulação. O conceito se baseia na divisão do modelo em partes com o objetivo de reduzir a da complexidade do modelo que está sendo simulado (CHWIF et al., 2006). Cada parte é implementada em um programa e alocada em um processo do Sistema Distribuído, de forma que as operações tenham condições de ser processadas paralelamente. Estes programas, contendo partes do modelo, são implementados em computadores distintos de

um Sistema Distribuído ou de uma máquina paralela. Porém, ao realizar tal implementação, vários obstáculos, que em um sistema centralizado não existiriam, surgem devido a este paralelismo, como: a sincronização dos processos, o balanceamento de carga e a sobrecarga na rede de comunicação.

Dentre estes problemas, um dos mais pesquisados é a sincronização dos processos, devido à dificuldade em coordenar os relógios físicos de forma confiável. Mesmo utilizando relógios de alta precisão e exatidão é impossível sincronizá-los, além do alto custo em adquirir estes componentes (TANENBAUM, 2003). A sincronização dos processos é necessária para que em uma simulação não ocorram erros de causa e efeito, que consistem em inconsistências relacionadas à ordem cronológica de processamento das tarefas. Em um sistema centralizado este tipo de erro não ocorre, pois todas as operações são processadas seqüencialmente. Já em sistemas distribuídos a relação de causa e efeito pode ser violada, pois as operações são processadas paralelamente podendo causar inconsistências nos resultados da simulação. Para que os resultados da simulação possam ser considerados confiáveis, a sincronização deve ser baseada na ordem cronológica do processamento de eventos e não em um relógio físico sincronizado (CHANDY & MISRA, 1979).

No decorrer dos anos, com a evolução dos estudos envolvendo sistemas distribuídos, duas classes de protocolos surgiram com o propósito de sincronizar e garantir a consistência dos resultados das simulações. A primeira classe é formada pelos protocolos conservativos e se caracterizam por não permitir a ocorrência de erros de causa e efeito. Enquanto que a segunda classe, formada pelos protocolos otimistas, permite a ocorrência dos erros de causa e efeito, vindo a tratá-los logo após a sua identificação (WANG & TROPPER, 2007).

Porém, mesmo com o desenvolvimento de tais protocolos, não é uma tarefa fácil aplicá-los em modelos para reduzir o tempo de simulação, isto devido à complexidade encontrada em suas estruturas. E, com intuito de permitir que estes modelos possam ser simulados em um ambiente distribuído por pesquisadores que não possuem conhecimento na área, a proposta deste

trabalho é definir e implementar um *framework* para o desenvolvimento de aplicações de simulação distribuída.

Conceituado por Johnson e Foote (1988), um *framework* se caracteriza por um conjunto de classes personalizadas e abstratas desenvolvidas para solucionar uma família de problemas relacionados, apresentando grande suporte para reutilização. Dentro desta visão, o desenvolvimento de um *framework*, contendo componentes que permitem realizar simulações em ambientes distribuídos, é viável e de grande utilidade.

1.3. Protocolos de Sincronização

Um ambiente distribuído pode ser representado por vários processos lógicos onde estão partes de um modelo matemático a ser simulado. Cada processo lógico possui variáveis de estado próprias que caracterizam parte dos resultados da simulação. Variáveis de estado de um processo lógico definem o *estado local do processo*. Enquanto que todos os estados locais juntos caracterizam o *estado global do sistema* (ZIMMERMANN, 2006).

A simulação avança de acordo com a criação, pelos processos, de eventos para serem executados. Estes eventos podem ser caracterizados por: um envio ou recebimento de uma mensagem ou por uma alteração nas variáveis de estado. Cada processo lógico pode criar eventos para si próprio ou para um dos demais processos do sistema, que deverá ser enviado através de uma mensagem. Por isto, cada processo possui uma lista de eventos em que todos os eventos recebidos ou criados, que devem ser processados durante a execução da simulação, são armazenados em ordem cronológica de execução. Esta ordenação é realizada através de uma marca de tempo, chamada *timestamp*, que todos os eventos possuem. Conforme a simulação avança o evento com o menor *timestamp* é retirado da lista e processado.

Como a simulação ocorre em vários computadores ao mesmo tempo, existe um mecanismo responsável pelo controle do progresso da simulação. Este controle é realizado por um *relógio global*. A simulação caminha de acordo

com a retirada dos eventos da lista de eventos futuros. Para cada evento retirado da lista, o relógio lógico de cada processo avança até o *timestamp* deste evento, dando assim o andamento da simulação. Como sempre o evento retirado da lista é o que possui o menor *timestamp* e, desta forma, é possível observar que a simulação é realizada em ordem cronológica.

Para que uma simulação distribuída tenha validade é necessário que não ocorram, durante a simulação, erros de causa e efeito, ou seja, a situação em que dois eventos e_1 e e_2, com o *timestamp* de e_1 maior que o *timestamp* de e_2, são processados na ordem inversa, ou seja, e_2 seja processado antes de e_1.

1.3.1. Os Protocolos Conservativos

Nesta seção, será realizada uma breve abordagem dos protocolos conservativos a fim de contextualizar a forma de como esta classe de protocolos se comporta na ocorrência de erros de causa e efeito. Os protocolos conservativos se caracterizam por evitar qualquer ocorrência de erros de causa e efeito. Assim um evento só pode ser tratado se o sistema garantir que nenhum evento com um *timestamp* menor do que o dele possa vir a ser processado no futuro. Para que exista esta garantia, o sistema *possui algumas características próprias:*

- ✓ Os canais de comunicação entre processos são estáticos, e desta forma, um processo só pode comunicar-se com outro processo se existir entre eles um canal definido antes do início da simulação;
- ✓ As mensagens que trafegam pelos canais de comunicação são armazenadas em fila mantendo sempre a ordem cronológica de execução;
- ✓ Toda mensagem que trafega entre os canais de comunicação deve estar rotulada com o relógio local *(LVT - Local Virtual Time)* do processo.

O grande desafio deste tipo de protocolo consiste em especificar quando é seguro executar um evento sem que ocorra um erro de causa e efeito. Quando não há a possibilidade de garantir esta segurança em nenhum dos

processos envolvidos na simulação, pode ocorrer uma situação de *deadlock*. Por esta razão, o tratamento de *deadlocks* em sistemas conservativos é de suma importância. Há duas linhas principais para o tratamento deste problema: uma pela qual o sistema deve evitar totalmente a ocorrência de *deadlocks*, e outra em que a sua ocorrência é permitida para, então, ser detectada e tratada.

Para prevenção da ocorrência de *deadlocks* uma técnica utilizada é a do envio de mensagens nulas para cada evento processado. Estas mensagens não possuem nenhum conteúdo que ocasione alguma atividade para o sistema, contém apenas o *LVT* do processo que a enviou e, assim, cada processo toma conhecimento dos *LVT's* dos processos que têm um canal de comunicação com ele, podendo identificar o menor *LVT* dentre eles.

O grande problema desta técnica é a sobrecarga nos canais de comunicação do sistema ocasionada pelas mensagens nulas. Para atenuar este problema, pode-se utilizar do envio de mensagens nulas sob demanda. Neste caso, não há envio de mensagens nulas sempre após o processamento de cada evento. O envio de mensagens nulas se dá após um *timeout* ou quando o canal que possui o menor relógio também possui uma fila vazia, indicando que o processo está bloqueado.

Em sistemas onde a ocorrência de *deadlocks* é muito rara, a utilização de mensagens nulas se torna inviável. Logo, a técnica utilizada é a de detecção e recuperação de *deadlocks* desenvolvida por Chandy e Misra (1981). Este protocolo possui um tipo especial de mensagem atuando como um marcador que percorre todos os canais da rede durante um ciclo. Neste sistema, cada processo também possui um *flag*, representado por um *bit*, que indica se o mesmo recebeu ou enviou uma mensagem desde a última passagem do marcador. Desta forma, são considerados dois tipos de processos: pretos e brancos. Os processos são brancos se o mesmo não recebeu nem enviou mensagens desde a última passagem do marcador, ao contrário do que ocorre quando o processo se torna preto.

Neste tipo de protocolo a ocorrência de *deadlock* é detectada quando o marcador ao passar por todos os canais da rede percebe que todos os processos estão brancos. O marcador também deve armazenar o número de

processos brancos encontrados desde o último processo preto para que ao detectar o *deadlock* o processo causador de *deadlock* possa ser reinicializado.

Outra otimização proposta por Misra (1986), que proporcionou uma redução considerável de tempo de execução do protocolo, foi a introdução de *lookahead*. Este conceito se caracteriza por um intervalo de tempo em que todos os processos não podem agendar eventos futuros. De acordo com Misra (1986), neste intervalo de tempo o processo tem certeza absoluta de que a execução destes eventos não irá causar erros de causa e efeito.

Nesta seção foram apresentadas algumas variações do protocolo *CMB* desenvolvido por Chandy, Misra (1979) e Bryant (1977), para auxiliar na compreensão do comportamento dos protocolos conservativos. Além das variações aqui apresentadas outras podem ser citadas para um estudo mais aprofundado deste tipo de protocolo, como: SRADS (REYNOLDS, 1982), *Appointments* (NICOL & REYNOLDS, 1984), *Turner Carrier-null Scheme* (CAI & TUNER, 1990) e *SPaDES/Java* (TEO & NG, 2002).

1.3.2. Os Protocolos Otimistas

Enquanto os protocolos conservativos previnem os erros de causa e efeito, os protocolos otimistas permitem aos processos executar eventos livremente sem limitações a fim de prevenir esse tipo de erros. Porém, se um erro ocorrer, os protocolos otimistas possuem mecanismos para recuperar o sistema e manter a consistência do mesmo. Para realizar a recuperação do sistema estes protocolos utilizam um mecanismo denominado *rollback*. Este mecanismo possui a função de retornar a computação a um ponto especificado para reprocessar os eventos corretamente e manter a consistência do sistema (WANG, 2007).

A sincronização dos processos e o controle cronológico do sistema dependem do gerenciamento dos registros empregados para este fim. Os registros utilizados são:

- ✓ O *Local Virtual Time (LVT)*, um contador de eventos que permite a cada processo acompanhar a ordem cronológica de sua computação. Este contador é incrementado toda vez que o processo

executa um evento, seja este proveniente de uma mensagem ou criado internamente pelo próprio processo.

- ✓ Outro registro de suma importância é o *timestamp*, já apresentado no capítulo anterior, que possui a função de indicar ao processo quando este evento, no qual o *timestamp* está inserido, deve ser executado.

- ✓ O último registro utilizado é denominado *Global Virtual Time (GVT)* que é representado pelo menor *LVT* do sistema, indicando que nenhum processo do sistema irá criar um evento com o *timestamp* menor do que o valor *GVT*.

Um erro de causa e efeito é identificado quando uma mensagem contendo um evento é recebida e possui um *timestamp* menor do que o *LVT* do processo receptor. Neste caso, o protocolo deve realizar um *rollback* para o *LVT* indicado no *timestamp* ou anterior para a recuperação do sistema.

Assim, ao realizar *rollbacks*, o sistema utiliza os valores encontrados nestes registros para identificar os erros de causa e efeito e encontrar o ponto exato que a computação deve retornar.

Nos próximos capítulos serão apresentados dois protocolos otimistas que fazem parte do escopo principal deste trabalho, o protocolo *Time Warp* e *Rollback* Solidário.

1.4. Estrutura da Dissertação

Após esta breve discussão sobre Simulação Distribuída, protocolos conservativos e otimistas, nos próximos capítulos serão apresentados a revisão de literatura realizada para se alcançar o resultado deste trabalho e todo o desenvolvimento do *framework* proposto.

Os capítulos dois e três descrevem os comportamentos dos protocolos otimistas *Time Warp*, e *Rollback* Solidário, desenvolvido por Moreira (2005), oferecidos como recursos do *framework*.

O capítulo quatro apresenta o estado da arte dos denominados *frameworks*, assim como sua classificação, as etapas de construção destes e alguns exemplos de ferramentas utilizadas como auxílio no desenvolvimento deste trabalho.

O capítulo cinco é o principal capítulo deste trabalho, pois, apresenta toda a modelagem e todo desenvolvimento do *framework* proposto. Todos os recursos e instruções de como utilizá-lo também estão contidos neste capítulo.

Por fim, o capítulo seis apresenta as discussões, conclusões referentes a este trabalho e sugestões para trabalhos futuros.

Capítulo 2: Protocolo *Time Warp*

As seções seguintes descrevem o comportamento do protocolo *Time Warp*, criado por Jefferson (1985) e sendo considerado referência entre os protocolos otimistas, por ter um grande número de artigos publicados sobre ele.

Como protocolo otimista, o *Time Warp* não impede a ocorrência de erros de causa e efeito. Esta característica faz com que todos os eventos sejam considerados seguros e executáveis. Desta forma, cada processo lógico executa seus eventos independentemente sem levar em consideração a situação da computação dos demais processos do sistema. Por isto, erros de causa e efeito só podem ocorrer através da troca de mensagens entre processos. Mensagens causadoras de erros são denominadas mensagens *stragglers*. Assim, quando uma destas mensagens é recebida, deve ser tratada através de procedimentos de *rollback* (ZIMMERMANN, 2006).

2.1. Inconsistências no Sistema

No protocolo *Time Warp* quando uma mensagem *straggler* é recebida o processo receptor deve realizar ações para reconstituir a computação, e torná-la novamente consistente. Dentre estas ações estão:

1. Identificar o LVT anterior ao *timestamp* da mensagem *straggler* recebida.

2. Retornar até o LVT identificado e processar o evento contido na mensagem *straggler*, realizando um *rollback*.

3. Verificar se alguma mensagem foi enviada durante este intervalo de retorno.

4. Em caso de positivo, o processo deve enviar novas mensagens para os processos receptores. Estas mensagens serão utilizadas para que os demais processos eliminem de sua “lista” as mensagens oriundas deste processo.

Mensagens que possuem a função de eliminar mensagens das listas dos processos envolvidos em um *rollback* são chamadas de anti-mensagens. Toda vez em que o processo realizar *rollback*, ele deve verificar se alguma mensagem enviada deverá ser eliminada. Constatada esta necessidade, o processo que está realizando *rollback* reenvia novamente esta mensagem. O processo receptor ao receber esta anti-mensagem terá duas mensagens iguais e de sinais opostos em sua lista de entrada, estas mensagens devem então ser eliminadas. Caso o processo receptor já tenha processado a mensagem em questão e enviado alguma mensagem neste período, o mesmo deverá realizar *rollback* e repetir o procedimento de envio de anti-mensagem descrito (ZENG, 2004). Os *rollbacks* gerados através de *anti-mensagens* são nomeados de *rollbacks secundários* (JEFFERSON, 1985; RAJAEI, 2007).

A figura 2.1 representa um Sistema Distribuído formado por três processos, representados por P_1, P_2 e P_3. As linhas horizontais representam a execução de um processo, com o tempo progredindo da esquerda para a direita, sendo que os círculos representam os eventos. As setas representam as mensagens que são trocadas entre os processos, sendo a base da seta um evento de envio e a ponta da seta o respectivo evento de recebimento da mensagem. Esta figura tem a função de exemplificar o comportamento do protocolo *Time Warp* durante o surgimento de uma mensagem *straggler*, evento e_{13}^{3}, no sistema.

Pode-se observar que a computação é consistente até o momento em que o processo P_3 recebe uma mensagem com *timestamp* igual a 15. Identifica-se na figura 2.1, que o último estado salvo ocorreu quando P_3 se encontrava com o *LVT* igual a 18. Esta mensagem provocará um erro de causa e efeito no sistema forçando o mesmo a reparar esta inconsistência. Então, o processo P_3 deverá realizar um *rollback* para o estado anterior ao *LVT* 15. Para isto, o processo P_3 terá que retroceder para o estado anterior ao evento e^3_{15}, ou seja, recuperar o estado salvo imediatamente após o evento e^3_{10}.

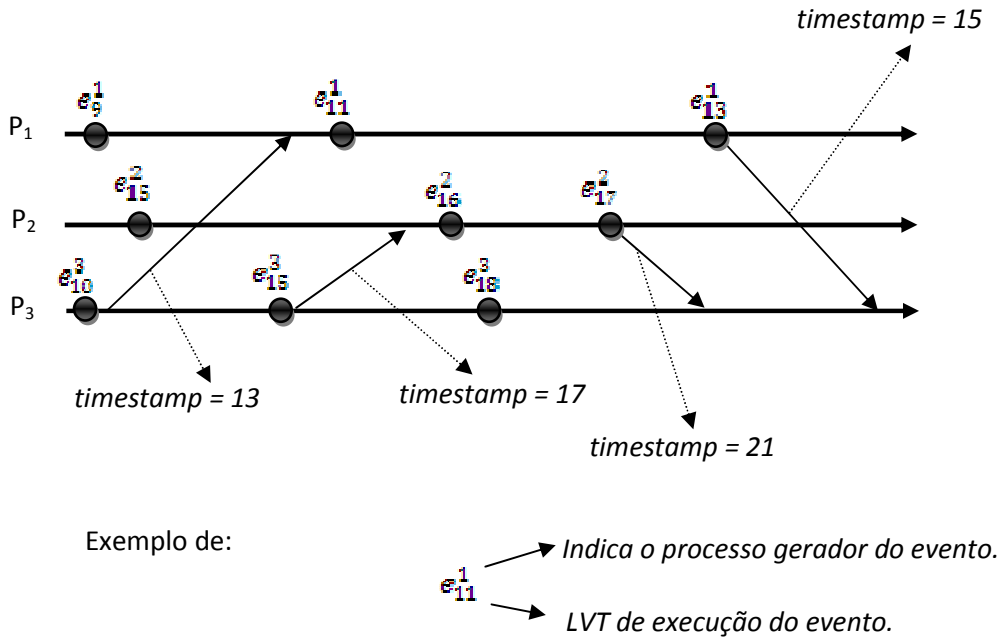

Figura 2.1: Funcionamento do Protocolo *Time Warp*.

Já o evento e^3_{15} havia gerado uma mensagem para o processo P_2 com um *timestamp* igual a 17, logo P_3 deverá enviar uma anti-mensagem para cancelar o evento gerado por ela. Ao cancelar o evento e^3_{17}, gerado pela mensagem enviada por P_3, P_2 também deverá cancelar a mensagem gerada por ele enviada a P_3 com um *timestamp* igual a 21, neste caso o evento será apenas descartado da lista de eventos futuros de P_3, pois este ainda não foi processado. Após todas estas ações a execução do sistema retorna ao procedimento normal de operação.

Para que este procedimento de recuperação do sistema consiga garantir as especificações cada processo utiliza o emprego de duas listas de armazenagem de eventos:

- ✓ a lista de eventos futuros, já citada anteriormente, em que são armazenados os eventos que devem ser tratados em um futuro próximo indicado pelos *timestamps* das mensagens recebidas;
- ✓ e a lista de anti-mensagens, onde todas as mensagens enviadas são armazenadas para serem utilizadas em caso de *rollback*.

Estas listas são empregadas, para garantir ao processo recursos necessários para a reconstituição do sistema de forma adequada e eficiente. Porém, há outros problemas que podem surgir no decorrer da execução dos eventos e devem ser tratados. *Rollbacks* em cascata podem provocar um grande prejuízo ao sistema, podendo fazer com que a execução dos eventos recue até o ponto inicial da computação (RAJAEI, 2007).

Outro fato que deve ser levado em consideração é a utilização de duas listas de armazenamento de dados. Com o andamento da computação estas listas podem ultrapassar os limites de armazenamento de dados do sistema, fazendo com que o mesmo entre em colapso.

2.2. Cálculo do *Global Virtual Time*

Uma maneira de prevenir o problema de estouro da memória é descartar eventos e anti-mensagens armazenados que não serão utilizados para uma recuperação de sistema.

Para garantir a confiabilidade do sistema, o protocolo utiliza um controle global a fim de gerenciar a memória do sistema e calcular o *Global Virtual Time (GVT)* (JEFFERSON, 1985). O *GVT* representa o menor *timestamp* entre os eventos internos ou externos que ainda não foram processados, de todos processos nas listas ou nas mensagens em trânsito. Se o *GVT* for igual ao menor *timestamp* de todos os processos do sistema, pode-se concluir que a

partir daquele ponto não haverá mais *rollbacks*. Porém, há uma limitação física que pode fazer com que o cálculo do *GVT* seja inconsistente, este problema ocorre devido ao atraso das mensagens durante o envio das mesmas na rede em que os computadores estão inseridos.

2.2.1. Mensagem Transiente

Nos protocolos otimistas não existe a garantia da ordem de chegada das mensagens enviadas. Logo uma mensagem pode sofrer atrasos durante seu percurso. Esta mensagem em trânsito, também conhecida como mensagem transiente, pode causar um grande problema durante o cálculo do *GVT* do sistema.

Em uma das abordagens do protocolo *Time Warp*, ao iniciar o processo de cálculo do *GVT,* cada processo deve enviar o valor de seu *LVT* para um processo controlador, e a partir do recebimento destes *LVT's* o processo controlador calcula o valor do *GVT* do sistema. Durante este procedimento uma mensagem que sofreu um atraso na rede pode possuir um *timestamp* menor do que o *GVT* calculado, fazendo com que o valor do *GVT* do sistema seja incorreto.

Na figura 2.2 o processo controlador envia para os demais processos uma mensagem solicitando os mínimos locais de cada um para realizar o cálculo do *GVT* do sistema. Os demais processos enviam respectivamente os valores 25, 24 e 26. Nota-se que o valor do *GVT* calculado será incorreto. Este fato ocorre devido ao atraso da mensagem gerada pelo evento e_{20}^{1} que não foi computada durante o procedimento do cálculo do *GVT*.

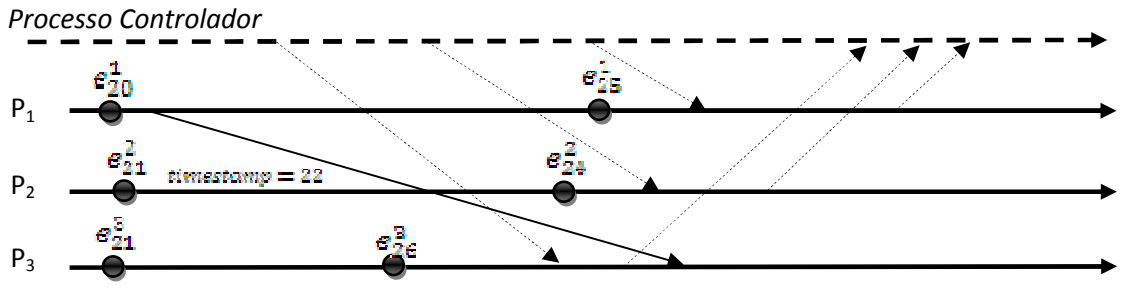

Figura 2.2: Mensagem Transiente

Um algoritmo desenvolvido por Fujimoto (2000) resolve o problema das mensagens transientes através de um sistema de confirmação de recebimento de mensagens. Neste algoritmo, o processo coordenador atua de forma ativa no cálculo do *GVT*, toda mensagem enviada deve ser confirmada pelo processo receptor. Assim, quando se inicia o cálculo do *GVT* do sistema, o processo coordenador envia uma mensagem em *broadcast* avisando processos que a partir daquele momento todos os processos interromper o processamento para dar início ao cálculo do *GVT* do sistema. Quando todos os processos confirmam o recebimento desta mensagem o processo coordenador envia outra mensagem para que os demais calculem o seu tempo lógico mínino.

Quando o processo coordenador receber as mensagens de todos os processos com os valores mínimos calculados, este enviará uma mensagem com o valor do *GVT* do sistema terminando, assim, o procedimento de cálculo do *GVT*. Pode-se observar que o processo responsável em contabilizar o tempo lógico da mensagem transiente é o processo que a enviou.

Este algoritmo, apesar de atender as especificações do sistema, limita o otimismo da simulação, isto ocorre, pois toda a computação é interrompida durante o cálculo do *GVT*. Uma forma eficiente de se realizar o cálculo seria a criação de um mecanismo em que não houvesse a necessidade de bloquear todo o sistema.

2.3. Algoritmo de Mattern

Vários algoritmos foram criados para permitir o cálculo do *GVT* sem o bloqueio da computação do sistema. A não interrupção do processamento pode provocar um problema conhecido como relatório simultâneo, que consiste na não inclusão dos tempos lógicos das mensagens transientes no cálculo do *GVT*, provocando erros.

Samadi (1985) solucionou o problema através de mensagens de confirmação. Assim, durante o processo de cálculo do *GVT,* as mensagens de

confirmação garantem que nenhuma mensagem em trânsito deixe de ser computada. Neste algoritmo, durante o cálculo do *GVT*, os *timestamps* das mensagens enviadas só são descartados pelos emissores após a confirmação do recebimento das mesmas pelos receptores. Desta forma, durante o procedimento de obtenção do *GVT*, a mensagem em trânsito na rede será computada pelo processo emissor.

Já o algoritmo de Mattern utiliza o conceito de corte consistente, que dividindo a computação em duas épocas: “passado” e “futuro”, para garantir que nenhuma mensagem deixe de ser computada durante a obtenção do *GVT* do sistema.

Um corte consistente é constituído por um subconjunto de eventos salvos, de maneira que, nenhum evento que antecede o corte seja precedido por um evento que sucede ao corte (ZIMMERMANN, 2006). Na figura 2.3 pode-se observar que o corte A não é um corte consistente, pois o evento e_{18}^{3} precede o evento e_{20}^{1}, não pertence ao corte. Por sua vez, o corte B é consistente, pois não existe nenhum evento após o corte que preceda outro antes do mesmo.

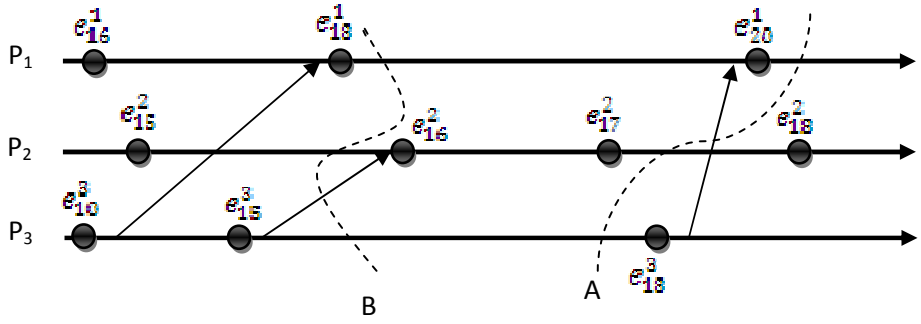

Figura 2.3: Representação de um corte consistente e ou não consistente.

Utilizando o corte consistente pode-se definir como *GVT*, de acordo com Mattern, o menor *LVT* ou *timestamp* do subconjunto dos eventos executados antes do corte consistente. Este procedimento permite ao processo coordenador obter o *GVT* do sistema de forma eficiente e consistente.

2.4. Salvamento de Estados

Os protocolos otimistas possuem várias vantagens em relação aos protocolos conservativos, porém necessitam de um mecanismo eficiente de salvamento de estados para que possam recuperar sua computação em caso da ocorrência de um erro de causa e efeito. Dentre estes mecanismos estão o *Copy State Saving* (JEFERSON, 1985), *Incremental State Saving* (STEINMAN, 1993b), *Sparse State Saving* (PREISS et al., 1992) (FLEISHMANN & WILSEY, 1995) e *Hybrid State Saving* (QUAGLIA & CORTELLESSA, 1997). Todos estes desenvolvidos para atender as necessidades de salvamento de estados dos protocolos otimistas como o protocolo *Time Warp*.

Neste texto, porém, serão apresentados apenas os mecanismos *Copy State Saving* e *Sparse State Saving*, devido a sua importância no desenvolvimento deste trabalho.

2.4.1. *Copy State Saving*

Como mecanismo mais simples de salvamento dos estados, o *Copy State Saving* se caracteriza por salvar todos os estados da Simulação Distribuída. Assim, quando há ocorrência de um erro de causa e efeito, a simulação retorna para o estado mais próximo e anterior ao tempo lógico do evento que deve ser recuperado.

Este método, apesar de simples, não é eficiente quando um sistema apresenta um baixo índice de *rollbacks.* Isto ocorre, pois o sistema perde muito tempo salvando estados que nunca serão utilizados. A figura 2.4 representa parte de uma simulação em que P_1 é um processo que faz parte de um sistema distribuído que utiliza o *Copy State Save* como seu mecanismo de salvamento de estados. As esferas representam eventos processados pelo sistema, em que os números subscritos representam o *LVT*, ou seja, o tempo lógico em que o evento é processado por P_1. As setas representam mensagens enviadas por outro processo e recebidas por P_1. Estas mensagens possuem eventos que deverão ser processados no *timestamp* anexado a ela. Neste exemplo, quando o processo recebe a mensagem contendo o evento e^2_{15}, que deverá ser

processado no *LVT* igual a 29, o processo P_1 irá inseri-lo em uma fila, denominada *lista de eventos futuros*. Esta lista contém todos os eventos que deverão ser processados em um tempo lógico futuro. Já o evento e_{19}^2 ao ser recebido pelo processo P_1, apresenta uma condição distinta do evento recebido anteriormente, o *timestamp* do evento e_{19}^2 é menor que o *LVT* atual do processo, neste caso será necessário que o processo P_1 realize um *rollback* para restaurar o sistema. Como o mecanismo de salvamento possui todos os estados salvos em sua memória, o processo realizará um *rollback* para o *LVT* imediatamente anterior ao *timestamp* do evento recebido.

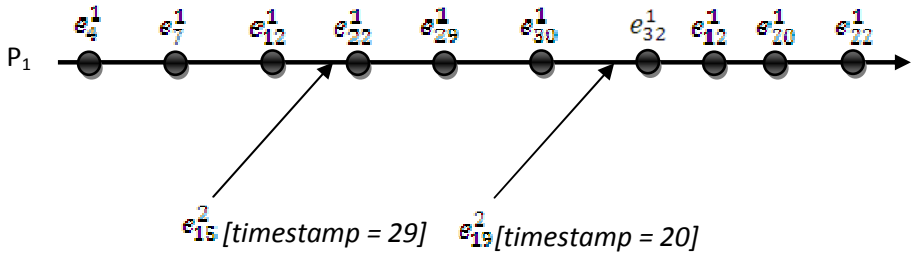

Figura 2.4: Funcionamento do mecanismo *Copy State Saving*

2.4.2. *Sparse State Saving*

O mecanismo *Sparse State Saving*, como uma evolução do mecanismo *Copy State Saving*, afim de não perder parte do tempo salvando estados que nunca serão utilizados, realiza uma forma distinta de salvamento. Neste mecanismo, os estados são salvos em um intervalo de iterações e não a cada evento executado. Outra vantagem adquirida pelo sistema é a diminuição da quantidade de memória necessária para sua operação. Desta forma, quando há um *rollback*, o estado recuperado será o estado salvo mais próximo do evento a ser processado, e não o imediato e anterior ao estado a ser recuperado.

Na figura 2.5 apenas os estados dos eventos marcados são salvos pelo mecanismo *SSS*, enquanto os demais são descartados. Assim, quando a mensagem *straggler* é recebida, o protocolo realiza *rollback* para o evento com *LVT* mais próximo e anterior aos eventos armazenados em sua memória, neste caso o evento e_7^1. A região de *coast forward*, apresentada na figura 2.4 persiste

do evento com *LVT* igual a 7 até o evento com *LVT* igual a 20, assim o processo não necessita reenviar nenhuma mensagem neste intervalo.

O intervalo de gravação dos estados pode ser programado para variar de acordo com a ocorrência de *rollbacks* da simulação. Logo, quando há uma ocorrência elevada de *rollbacks*, o intervalo de gravação de estados deverá ser menor e o mecanismo *Sparse State Saving* se parecerá muito com o mecanismo *Copy State Saving.* Porém, quando o sistema apresentar uma menor incidência de *rollbacks*, o mecanismo *Sparse State Saving* se torna muito eficiente, pois o tempo perdido no salvamento de estados que não serão utilizados será significativamente pequeno.

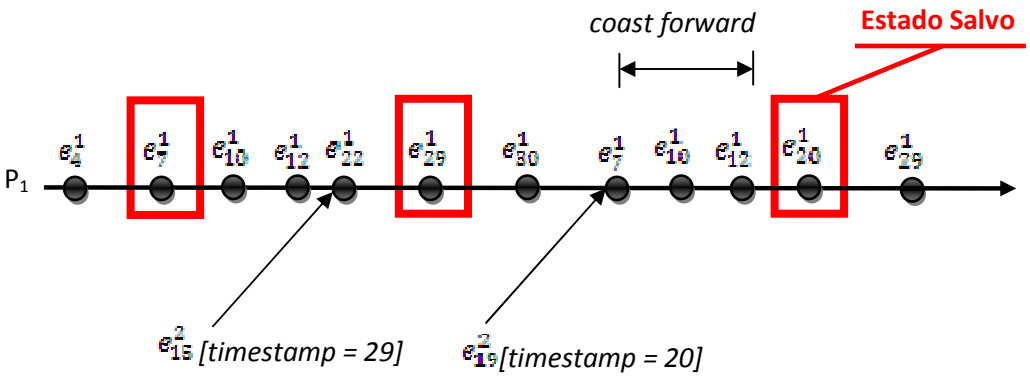

Figura 2.5: Funcionamento do mecanismo *Sparse State Saving*

2.5. Anti-Mensagens

Como já discutido em seções anteriores, as anti-mensagens são de suma importância para o funcionamento do protocolo *Time Warp*, pois são responsáveis em garantir a consistência da computação quando um erro de causa e efeito ocorre em algum processo. Porém, assim como os eventos comuns, as anti-mensagens ocupam espaço na memória e necessitam ser eliminadas quando não serão utilizadas.

O mecanismo responsável em eliminar as mensagens armazenadas na lista de anti-mensagens é o mesmo mecanismo do cálculo do *GVT*. Assim,

todas as anti-mensagens que possuam um *timestamp* menor que o *GVT* calculado podem ser eliminadas. Isto é possível devido à garantia dada pelo mecanismo de controle do *GVT* em que não há possibilidade de ocorrer um erro de causa efeito para qualquer evento com *LVT* menor do que o último *GVT* calculado.

Outro fator importante que deve ser considerado na implementação do protocolo é como o sistema deve se comportar na necessidade do envio de anti-mensagens. Neste caso, existem três estratégias utilizadas para o cancelamento dos eventos processados ou não durante a recuperação do sistema (GAFNI, 1988; REIHER et al., 1990; ISKRA et al., 2003):

- ✓ **Cancelamento Agressivo**: em que todas anti-mensagens são enviadas imediatamente para os demais processos que receberam mensagens do processo que realiza o *rollback*.
- ✓ **Cancelamento Preguiçoso**: em que as anti-mensagens só são enviadas se a nova execução da computação, após o procedimento de *rollback*, não coincide com a computação anterior.
- ✓ **Cancelamento Dinâmico:** em que o sistema decide qual das estratégias de cancelamento deve ser utilizada.

As anti-mensagens no protocolo *Time Warp* possuem a mesma importância dos eventos comuns, pois sem elas não é possível restaurar o sistema em caso de erros de causa e efeito.

2.6. Considerações Finais

As utilizações de *rollbacks* com intuito de recuperar sistemas inconsistentes permitem aos protocolos otimistas grande liberdade na execução de eventos, proporcionando um alto ganho em velocidade na obtenção de resultados em simulações. Porém, em sistemas em que a ocorrência de *rollbacks* é muito alta os protocolos conservativos podem ser aplicados com grande eficiência.

A grande desvantagem de um protocolo como o *Time Warp* é a necessidade de armazenamento de anti-mensagens que ocupam um grande espaço na memória do sistema. Assim, o surgimento de um protocolo que não utiliza anti-mensagens pode ser bastante viável. O surgimento do protocolo *Rollback Solidário*, proposto por Moreira em 2005, apresenta uma nova solução utilizando o conceito de *checkpoint*s globais consistentes para recuperação do sistema.

Capítulo 3: *Rollback* Solidário

O protocolo *Rollback Solidário*, assim como o protocolo *Time Warp,* também utiliza *rollbacks* para recuperar estados quando um erro de causa e efeito ocorre. Esta e outras características fazem que ele pertença à classe de protocolos otimistas, porém, há diferenças significativas que não transformam o protocolo *Rollback Solidário* em uma variante do protocolo *Time Warp.*

Moreira (2005) propôs um novo protocolo que não necessita de anti-mensagens durante a recuperação do sistema, e para isto, utiliza o conceito de *checkpoints* globais consistentes (BABAOGLU & MARZULLO, 1993). A principal diferença entre os protocolos *Time Warp* e *Rollback Solidário* está na forma como cada um trata uma mensagem *straggler* quando esta é gerada durante uma simulação.

No protocolo *Time Warp*, quando uma mensagem *straggler* é recebida, o processo que a identificou realiza *rollback* imediatamente, para que o sistema se mantenha consistente. Este processo, ao recuperar sua computação, deve identificar todas as mensagens que foram enviadas durante este período e enviar anti-mensagens para que os processos que receberam alguma mensagem durante este intervalo também corrijam seus estados.

Já, no protocolo *Rollback* Solidário, quando uma mensagem *straggler* é identificada, o sistema faz um levantamento para identificar quais processos devem realizar um retrocesso, e assim, todos estes simultaneamente realizam *rollback*. Para isto, o protocolo *Rollback* Solidário utiliza-se da teoria dos

Checkpoints Globais Consistentes para identificar o ponto de recuperação em que os processos devem realizar a recuperação do sistema.

Com a realização de *rollbacks* em conjunto para um *checkpoint* global consistente não existe a necessidade de o sistema utilizar anti-mensagens, desta forma, os processos não necessitam armazenar cópias das mensagens que foram trocadas entre si, economizando, assim, memória do sistema. Outra vantagem considerável com a eliminação das anti-mensagens é a diminuição do tráfego na rede de comunicação, pois, como o ponto de retorno é enviado em *broadcast* pelo processo observador para os demais processos, não há anti-mensagens ocupando os canais de comunicação do sistema (MOREIRA, 2005). Os *rollbacks solidários* também garantem a não ocorrência de *rollbacks* em cascata, pois todos os processos envolvidos na recuperação do sistema retornam para um *checkpoint* global consistente. Como os estados salvos são concorrentes, não existe a possibilidade de um *rollback* gerar outro *rollback* e assim por diante. A utilização de *checkpoints* globais consistentes também permite que haja uma simplificação no cálculo do *Global Virtual Time* (*GVT*) do sistema.

Já o mecanismo de salvamento de estados do protocolo se assemelha com o mecanismo *Sparse State Saving* utilizado pelo protocolo *Time Warp*. No protocolo *Rollback* Solidário há salvamentos de duas formas distintas: em intervalos de tempo estipulados ou em situações “especiais” que provocam ocorrência de *checkpoints* forçados, utilizados para garantir a consistência do protocolo. Estes *checkpoints* forçados são gerados a fim de permitir que todos os *checkpoints* pertençam a pelo menos um *Checkpoint* Global Consistente.

3.1. Mecanismo de Salvamento de Estados e Funcionamento do Protocolo *Rollback* Solidário

A principal característica dos protocolos otimistas é tratar erros de causa e efeito ao invés de preveni-los e o quê os diferencia entre si é a forma de

realizar tal tratamento. Características como: mecanismo de salvamento de estados, tratamento de mensagem *straggler*, obtenção do *GVT* e sincronização do sistema, podem ser utilizadas para comparar protocolos otimistas distintos, pois interferem expressivamente na eficiência dos protocolos do sistema distribuído em questão.

3.1.1. Mecanismo de Salvamento do *Rollback* Solidário

Como o protocolo *Rollback* Solidário utiliza-se de *checkpoints* para recuperar a consistência do sistema, o mecanismo de salvamento de estados deste protocolo deve garantir que todo *checkpoint* local realizado faça parte de pelo menos um *checkpoint* global consistente, para que não haja desperdício de memória e tempo de simulação. Para isto, o protocolo utiliza-se de dois tipos de *checkpoints*: básicos e forçados.

O salvamento dos *checkpoints* básicos é semelhante ao mecanismo *Sparse State Save*, utilizado pelo protocolo *Time Warp,* em que os estados são salvos em um intervalo determinado de tempo. Este intervalo pode ser implementado levando em consideração a ocorrência de *rollbacks* durante a simulação, sendo maior quando a ocorrência de *rollbacks* for baixa ou menor quando a ocorrência de *rollbacks* for alta.

Já o salvamento de *checkpoints* forçados é baseado no algoritmo *Fixed Dependency After Send* (*FDAS*). Este algoritmo, implementa o padrão *No Receive After Send* (*NRAS*), proposto por Wang em 1997, que garante que todos os *checkpoints* locais façam parte de pelo menos um *checkpoint* global consistente. Isto é possível eliminando todos os caminhos-Z não causais entre quaisquer dois *checkpoints,* não necessariamente distintos. Caminhos-Z ou caminhos *zigzag* é uma seqüência de mensagens entre um subconjunto de *checkpoints* locais que não permitem a formação de *checkpoints* globais consistentes, isto pode ocorrer quando um processo lógico envia e recebe mensagens necessariamente nesta ordem no mesmo intervalo de *checkpoints* (NETZER & XU, 1995). O padrão *NRAS* pode ser observado na figura 3.1.

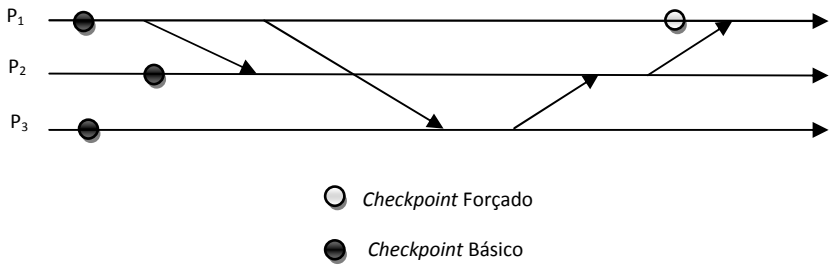

Figura 3.1: Padrão de *checkpoints* do padrão *NRAS*

Outra importante característica encontrada no algoritmo *FDAS* é o fato dele implementar o padrão de *checkpoints* que satisfaz a propriedade *Rollback-Dependency Trackability* (*RDT*). Esta propriedade é importante, pois permite rastrear todas as dependências entre *checkpoints* em tempo de execução através da utilização de vetores de dependências (WANG, 1997). Já a utilização de vetores de dependências facilita a identificação de um *checkpoint* global consistente específico, durante alguma recuperação do sistema.

3.1.2. Mensagem *Straggler*

Assim como nos demais protocolos otimistas, no protocolo *Rollback* Solidário uma mensagem *straggler* surge quando um processo recebe uma mensagem com *timestamp* menor do que o seu *LVT*. Entretanto, o protocolo realiza o tratamento da seguinte forma:

- ✓ Identifica em seus *checkpoints* salvos aquele que possui um tempo lógico imediatamente anterior ao *timestamp* da mensagem recebida;
- ✓ Retorna para o *checkpoint* identificado;
- ✓ Envia a informação a respeito do *rollback* e ponto de recuperação para o processo observador;
- ✓ O processo observador identifica o *checkpoint* global consistente que contenha o *checkpoint* local para o qual o processo retornou e que cause o menor prejuízo possível à computação;

- ✓ E envia para todos os processos uma mensagem contendo o *checkpoint* global consistente para o qual todos os demais processos devem retornar;
- ✓ Os demais processos devem atualizar seu *checkpoint* local e realizar *rollback* para o tempo lógico indicado na mensagem recebida e,
- ✓ A partir deste ponto o sistema retoma a computação.

O *checkpoint* global consistente em que o sistema deve retornar é formado por vários *checkpoints* locais, sendo um para cada processo. Porém, um *checkpoint* global pode possuir vários *checkpoints* locais de um mesmo processo, então o sistema deve utilizar aquele que causar o menor prejuízo para a simulação.

Cada *checkpoint* global consistente identificado pelo sistema pode ser denominado como linha de recuperação. Assim, no decorrer da simulação, o sistema irá possuir um conjunto de linhas de recuperação que serão utilizadas, caso necessário, para recuperar a simulação. Em caso de *rollback*, o sistema irá possuir um conjunto de linhas de recuperação que poderão restaurar o sistema de forma consistente. A única diferença entre elas será o custo da recuperação para a simulação. No diagrama representado na figura 3.2, destaca-se três linhas de recuperação: *A*, *B* e *C*. Qualquer uma delas pode ser utilizada para restaurar o sistema quando a mensagem *straggler* for recebida pelo processo P_1. Porém, a linha de recuperação *C* apresenta um menor custo para a simulação. Este custo pode ser definido como o intervalo entre o ponto em que o processo deveria retornar e o *checkpoint* global empregado para a recuperação do sistema. Comparando estes intervalos pode-se medir o prejuízo provocado por cada linha de recuperação apresentada, sendo que C_1, C_2 e C_3 são os custos de cada linha de recuperação, caso o sistema venha a retornar para qualquer uma destas. Na figura 3.2 o custo de cada linha de recuperação possui:

$$C_1 < C_2 < C_3$$

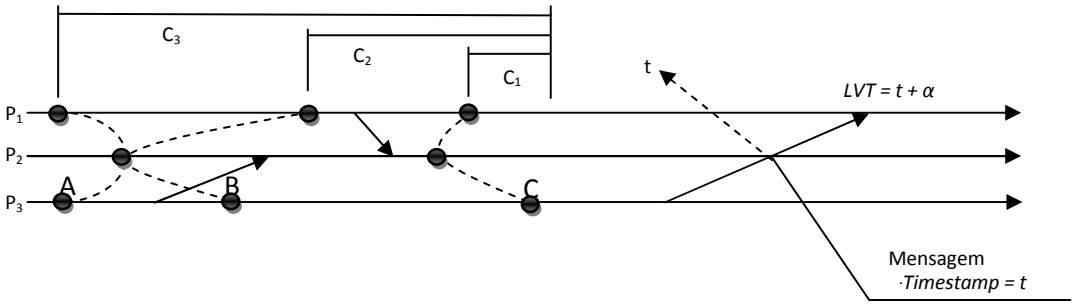

Figura 3.2: Linhas de Recuperação

De acordo com o comprimento de cada *custo,* pode-se classificar as linhas de recuperação em máxima e mínima, sendo:

- ✓ Linha de recuperação máxima: aquela que causa o menor prejuízo para a simulação, representado pela linha de recuperação *C* da figura 3.2,
- ✓ Linha de recuperação mínima: formada pelo *checkpoint* global consistente que produz o maior prejuízo para o sistema em caso de *rollback*, indicado pela linha de recuperação *A* da figura 3.2.

3.2. Abordagens do Protocolo

Na implementação do protocolo *Rollback* Solidário pode-se utilizar dois diferentes mecanismos para a obtenção de *checkpoints* globais consistentes, que influenciam diretamente no desempenho da simulação. O mais simples de ser implementado é utilizar um mecanismo síncrono para obtenção de *checkpoints*, que possui a vantagem de otimizar a memória do sistema, porém, sofre com o tempo perdido durante a obtenção dos *checkpoints* globais consistentes. A outra implementação emprega um mecanismo semi-síncrono que não possui a necessidade de parar o sistema para obter os *checkpoints* globais consistentes, e sua desvantagem é a utilização de uma quantidade maior de memória de armazenamento. Em ambos os casos pode-se obter *checkpoints* globais consistentes de forma confiável para uma eventual recuperação do sistema.

3.3. Mecanismo de Obtenção de *Checkpoint* Semi-Síncrono

Na abordagem semi-síncrona há um processo observador que possui uma função passiva durante a simulação, atuando como coordenador apenas quando o sistema necessita realizar um procedimento de *rollback*.

Todos os processos possuem um vetor de dependências em que cada posição representa um processo do sistema. Assim, se o sistema é formado por *n* processos, cada um destes possui um vetor de dependências (VD_n) com *n* posições. Este vetor de dependências possui a função de identificar a relação de precedência causal entre os *checkpoints* do sistema. Inicialmente, o vetor possui o valor *1* na posição do processo e zero nas demais posições, por exemplo, um vetor de dependências do processo P_2 inicia-se com VD_2 = *{0,1,0,...}*. Quando um processo realiza um *checkpoint* este deve incrementar sua posição no vetor de dependências, então, se o processo P_2 realizar um novo *checkpoint*, seu VD_2 será igual a *{0, 2, 0,...}*.

Quando uma mensagem é enviada para outro processo, junto a ela deve ser anexado seu vetor de dependências, para que o processo receptor, ao recebê-la, possa atualizar seu vetor de dependências. A atualização do vetor ocorre da seguinte forma:

1. O processo compara as posições do vetor de dependências recebido com os valores do seu próprio vetor, substituindo cada posição pelo maior valor entre eles.

2. Se o processo recebeu informação de um novo *checkpoint* e ele havia enviado uma mensagem após seu último *checkpoint* local, ele realizará um *checkpoint* forçado.

Desta forma, se o processo P_2 com VD_2 = *{1,3,2,1}* receber uma mensagem do processo P_3 contendo VD_3 = *{0,2,3,1}*, caso o mesmo ao ter recebido esta mensagem realizou um *checkpoint* forçado, após a atualização, o valor de VD_2 do processo P_2 será igual a *{1,4,3,1}.* Caso não tenha sido

necessário realizar um *checkpoint* forçado, o valor de VD_2 do processo P_2 será igual a *{1,3,3,1}.*

Junto à mensagem enviada ao processo observador contendo o vetor de dependências, cada processo deve anexar outro vetor contendo os *LVT's* dos respectivos *checkpoints* locais armazenados. Isto deve ocorrer também quando os processos trocam mensagens entre si. O vetor contendo os *LVT's* é necessário para que os processos receptores possam eliminar ou não uma mensagem armazenada na lista de eventos futuros durante um procedimento de *rollback*.

Diferentemente dos demais processos, que ao receberem um vetor de dependências anexado a uma mensagem atualizam seu vetor de dependências, o processo observador atualiza uma matriz quadrada de ordem *n*, em que *n* é o número de processos do sistema. Esta matriz é utilizada para que o processo observador identifique a linha de recuperação em caso de erros de causa e efeito.

Quando um processo envia seu vetor de dependências para o processo observador, este atualiza sua matriz colocando os valores do vetor de dependências na linha correspondente àquele processo. Logo, o vetor de dependências enviado pelo processo P_i ocupará a linha *i* da matriz do processo observador, já as outras posições da linha são ocupadas pela relação de dependências entre o processo P_i e os outros processos da simulação.

Os vetores de dependências enviados pelos demais processos também podem ser armazenados pelo processo observador em listas encadeadas. Neste caso, o processo observador mantém uma lista para cada processo do sistema. Em ambos os casos, em uma matriz ou em uma lista encadeada, o processo observador consegue obter os *checkpoints* globais consistentes de forma rápida e simples.

A principal função do processo observador é adquirir linhas de recuperação para serem utilizadas durante os procedimentos de *rollback* do sistema. Como na abordagem semi-síncrona não há interrupção da computação para a obtenção dos *checkpoints* globais consistentes, os demais

processos enviam ao processo observador mensagens contendo seus *checkpoints* locais para que o processo observador possa identificar e salvar as linhas de recuperação do sistema. Estas mensagens, contendo os *checkpoints* locais de cada processo, são enviadas em intervalos específicos para o processo observador durante a computação pelos demais processos do sistema.

Quando os vetores de dependências são armazenados pelo processo observador em uma matriz quadrada, pode-se obter, na diagonal principal da mesma, o último *checkpoint* local recebido pelo processo observador de cada processo. Assim, em um sistema que possui três processos, as posições M_{11}, M_{22} e M_{33} são formadas pelos *checkpoints* mais atuais de cada processo.

Um *checkpoint* global consistente é formado por *checkpoints* locais que não possuem relação causal entre si. Desta forma, o processo observador pode identificar um *checkpoint* global consistente comparando os valores da diagonal principal com os outros valores das respectivas colunas da matriz.

A figura 3.3 representa um trecho da simulação de um sistema que possui três processos e um processo observador. Inicialmente, cada processo envia para o processo observador uma mensagem contendo o vetor de dependências indicando o início da simulação.

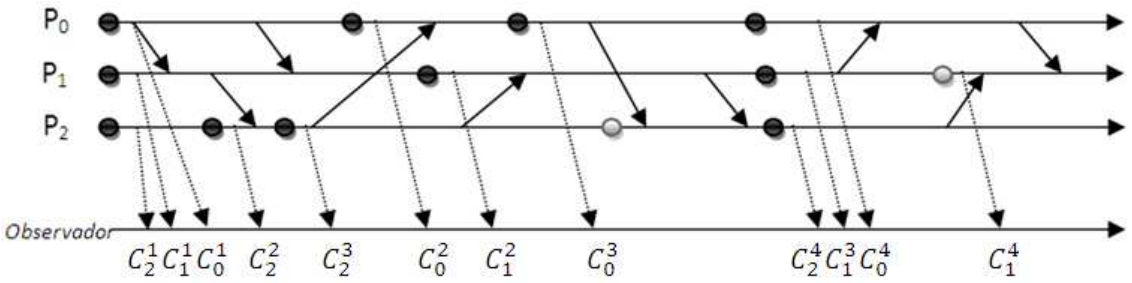

Figura 3.3: Comportamento do Protocolo *Rollback* Solidário Abordagem Semi-síncrona

Assim, a primeira matriz formada pelo recebimento destas mensagens será sempre uma matriz identidade:

$$M = \begin{Bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{Bmatrix}$$

A matriz identidade, além de indicar o início da simulação, também marca o primeiro *checkpoint* global consistente da simulação, formado pelo vetor *{1,1,1}*. Por conseguinte, quando o processo observador recebe a mensagem contendo o terceiro *checkpoint* do processo P_3 o valor da matriz será:

$$M = \begin{Bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 3 \end{Bmatrix}$$

Neste caso, a diagonal principal ainda não representa um *checkpoint* global consistente devido ao fato de que os valores de M_{11} e M_{22} são iguais aos valores de M_{31} e M_{32}. Por conseguinte, pode-se concluir que há relação causal entre estes *checkpoints*. Neste momento, não é possível que o processo observador obtenha um *checkpoint* global consistente, por isto, ele aguarda o recebimento de mais mensagens contendo os vetores de dependências dos demais processos. O segundo *checkpoint* global consistente, após o início da simulação, será obtido quando o processo observador receber o *checkpoint* local C_0^3, formando a matriz:

$$M = \begin{Bmatrix} 2 & 0 & 0 \\ 1 & 2 & 0 \\ 1 & 1 & 3 \end{Bmatrix}$$

Observando a matriz pode-se perceber que todos os valores posicionados na diagonal principal são maiores que os valores de suas colunas correspondentes, caracterizando o *checkpoint* global consistente formado pelo vetor *{2,2,3}*.

Apesar de eficiente na obtenção de *checkpoints* globais consistentes, esta abordagem utilizando matrizes, não proporciona a obtenção de todos os *checkpoints* globais consistentes possíveis. Isto ocorre, pois toda vez que o processo observador recebe uma mensagem contendo um *checkpoint* local, este substitui a linha contendo o *checkpoint* anterior pelo vetor de dependências do novo *checkpoint* local recebido. Por exemplo, na figura 3.3 o *checkpoint* global consistente formado pelo vetor *{2,2,2}* não foi obtido.

A utilização deste mecanismo para obter *checkpoints*, em caso de *rollback* pode fazer com que a simulação retorne para um intervalo maior do que o necessário, comprometendo a eficiência da simulação em si.

Para solucionar este problema o processo observador pode armazenar os *checkpoints* locais em listas encadeadas. E, quando existe a necessidade de obtenção de algum *checkpoint* global consistente, o processo observador percorre toda a lista encadeada buscando o *checkpoint* global consistente que oferece o menor prejuízo possível para computação. Desta forma, nenhum *checkpoint* é descartado indevidamente, a única deficiência em relação a abordagem utilizando matrizes, é que o processo observador necessita de um maior período de tempo para localizar o melhor *checkpoint* global consistente.

Como discutido anteriormente, o processo observador mantêm uma lista encadeada para cada processo contendo todos os *checkpoints* locais recebidos. Assim, quando o processo observador necessita encontrar um *checkpoint* global consistente para recuperar o sistema, o mesmo percorre todas as demais listas para encontrar a linha de recuperação máxima, causando deste modo, o menor prejuízo ao sistema.

Dos *checkpoints* recebidos pelo processo observador, no exemplo apresentado na figura 3.3, as listas encadeadas construídas são:

P_0	*1 0 0*	*2 0 0*	*3 1 3*	*3 1 4*	
P_1	*0 1 0*	*1 2 0*	*1 3 3*	*1 4 3*	
P_2	*0 0 1*	*0 0 2*	*1 1 3*	*1 1 4*	*3 2 5*

Figura 3.4: Listas encadeadas armazenada pelo processo observador

No exemplo representado na figura 3.3, caso o processo P_1 necessite voltar ao *checkpoint* local *{1,2,0}*, o processo observador irá comparar este *checkpoint* com todos os demais *checkpoints* locais armazenados nas listas encadeadas dos outros processos, e deve identificar todos os *checkpoints* que

não possuam precedência causal com este *checkpoint*. Observando as listas encadeadas representadas na figura 3.4, os *checkpoints* locais que não possuem relação causal com o *checkpoint {1,2,0}* são:

- ✓ Para o processo P_0: *{2,0,0}*; e
- ✓ Para o processo P_2: *{1,1,3}* e *{1,1,4}*.

Note que nenhum dos *checkpoints* encontrados possui um valor maior ou igual a dois na posição do processo P_1, indicando que nenhum destes *checkpoints* possui precedência causal com o *checkpoint {1,2,0}*. No entanto, o processo observador avalia a dependência entre todos os *checkpoints* candidatos a formar um conjunto consistente, para verificar se existe relação causal entre eles. Por exemplo, o *checkpoint* P_2 *{1,1,3}* seria desconsiderado após encontrar todos os *checkpoints* globais consistentes o processo observador escolherá o melhor *checkpoint* para a realização do *rollback* solidário.

Observando as listas encadeadas representadas na figura 3.4 é possível obter vários *checkpoints* globais consistentes combinando-os e utilizando a regra apresentada anteriormente. Os seguintes *checkpoints* globais consistentes podem ser obtidos através das listas da figura 3.4: *{1,1,1}, {2,1,1}, {2,1,2}, {1,1,2}, {2,2,2}, {2,2,3}, {2,3,4}, {3,2,4}, {3,3,4}, {4,2,4}, {4,3,4}, {3,4,4}, {2,4,4}, {4,4,4}, {4,3,5} e {4,4,5}.*

Já os *checkpoints* globais consistentes obtidos pelo método da matriz quadrada seriam apenas: *{1,1,1}, {1,1,2}, {2,2,3}, {3,2,3}, {3,3,5}, {3,3,5}.*

No exemplo aqui apresentado pode-se perceber que o método utilizando listas encadeadas oferece mais recurso para o sistema, isto ocorre devido ao fato de que o processo observador possui uma quantidade maior de *checkpoints* locais armazenados.

3.3.1. Tratamento das Mensagens *Straggler* Utilizando a Abordagem Semi-Síncrona

Assim como na abordagem síncrona, o sistema necessita de um processo coordenador para realizar o *rollback solidário* quando um processo recebe uma mensagem *straggler*. Neste caso, o processo responsável em coordenar o sistema durante um *rollback* é o processo observador, que até este momento possuía uma função passiva durante a simulação.

Ao identificar uma mensagem *straggler*, o processo que a recebeu irá iniciar o procedimento de *rollback* através de uma mensagem para o observador, indicando que o mesmo está realizando um procedimento de *rollback.* O *checkpoint* local, para onde o processo irá retornar, é o conteúdo principal da mensagem. Neste momento, o processo observador identifica a linha de recuperação ideal para a restauração do sistema. Após a identificação do *checkpoint* global consistente o processo observador, agora na função de coordenador, envia uma mensagem contendo o *checkpoint* de retorno para todos os processos da simulação, inclusive o processo que iniciou o procedimento de *rollback*.

O processo que recebeu a mensagem *straggler* já restaurou o sistema, portanto, fica aguardando a mensagem de confirmação do processo observador, contendo o *checkpoint* global consistente, para atualizar seu estado e continuar a simulação. Já os demais processos, ao receberem a mensagem contendo o *checkpoint* de retorno, restauram o sistema com a linha de recuperação enviada pelo processo observador, salvam o estado recuperado e retornam para simulação.

3.4. Aspectos Gerais do Protocolo *Rollback* Solidário

Assim também como os protocolos *Time Warp* e *CMB*, o protocolo *Rollback* Solidário possui algumas características que interferem diretamente em seu desempenho. Algumas delas já foram discutidas nos capítulos anteriores como:

- ✓ A facilidade no cálculo do GVT,
- ✓ A não ocorrência de *rollback* em cascata,
- ✓ Uma melhor utilização da memória,
- ✓ E a otimização da rede de comunicação.

Porém, há alguns problemas que podem surgir devido à forma como o protocolo é implementado, ou seja, o próprio comportamento durante o tratamento das mensagens *stragglers* pode permitir que ocorram algumas situações que afetam a eficiência do sistema durante a simulação. Estas situações surgem quando:

- ✓ Um processo ao enviar uma mensagem para outro processo, para ser tratada em um *timestamp* futuro, sofre um *rollback* para um tempo anterior a criação da mensagem, e o processo receptor recebe este evento após a conclusão de todo procedimento de recuperação do sistema. Neste caso, o processo receptor tratará uma mensagem que deveria ser eliminada, este fato é conhecido como *Problema da Mensagem Transiente*.
- ✓ Um processo é obrigado a retornar para uma linha de recuperação, indicada pelo observador, mesmo que este não possua relação de dependência causal com o processo que necessariamente realizou *rollback*, este fato é denominado como *Anomalia do Retorno Desnecessário*.

- ✓ Todos os processos retornam para a última linha de recuperação identificada pelo processo observador e o evento que provocou o *rollback* deixa de existir. Assim, este evento poderia ser gerado novamente, fazendo com que a simulação entrasse em *livelock.*

- ✓ Surgem duas ou mais solicitações de *rollback* simultaneamente. Desta forma, o processo observador não possui condições de identificar se o pedido tratado primeiramente abrange ou não os outros pedidos da fila.

Para solucionar estes problemas a utilização de épocas se torna muito útil, pois todo o processo do sistema, inclusive o processo observador, pode identificar “a validade” de um evento ou de uma solicitação de *rollback* recebida.

Neste tipo de aplicação, o conceito de épocas é empregado para verificar se uma mensagem recebida não interfere na consistência do sistema. Mas, para que isto seja possível, algumas ações devem ser sempre realizadas por todos os processos da simulação, inclusive o processo observador.

3.4.1. Tratando a Anomalia do Retorno Desnecessário

Apesar de não fazer com que o sistema se torne inconsistente, o problema de Anomalia do Retorno Desnecessário pode prejudicar o desempenho do programa de simulação, pois, cada vez que um processo retorna sem necessidade, o processamento já realizado após o ponto de retorno é perdido.

Uma forma de solucionar este problema é fazer com que cada processo, ao receber uma mensagem do processo observador indicando um ponto de retorno, verifique se deve retornar ao ponto indicado ou não. Mas para isto, o processo observador deve enviar, anexado a mensagem indicando o *checkpoint* de retorno, a informação de qual processo iniciou o *rollback*. Assim, quando um processo recebe uma mensagem contendo um pedido de retorno, este deve verificar se o processo causador do *rollback* possui relação causal com ele e tomar a decisão de recuperar ou não o sistema.

Quando um processo identifica que não possui relação causal com o processo que iniciou o procedimento de *rollback*, este deve realizar um *checkpoint* e continuar a simulação normalmente.

3.4.2. Utilizando Épocas para Manter a Consistência do Sistema

Quando se aplica o conceito de época em uma simulação, esta pode ser separada em dois períodos: passado e presente. Inicialmente, todos os processos partem de uma época de valor ordinal 1 e, de acordo com a necessidade do sistema, este número irá sendo incrementado para o controle presente da simulação.

Esta época pode ser incrementada de várias formas, uma delas é na atualização do *GVT* da simulação. Logo, toda vez que se calcula o *GVT* do sistema todos os processos são informados para atualizarem também a sua época. Outra forma de atualização ocorre quando um procedimento de *rollback* é concluído. Neste momento, o processo observador cria uma nova época e envia este valor juntamente com a mensagem contendo a linha de recuperação do sistema.

Desta forma, todas as mensagens podem ser auditadas pelo processo receptor para que não haja inconsistência. Se um processo recebe uma mensagem de época diferente da sua, ou seja, uma mensagem do passado, este verifica se deve descartá-la, pois esta será provavelmente uma mensagem transiente que não foi considerada durante uma recuperação de sistema.

Outra situação que pode ser solucionada com a utilização de épocas é o problema com os *rollbacks* simultâneos. Neste caso, o processo observador armazenará todos os pedidos de *rollback* em uma fila e tratará de cada um deles, dependendo das mensagens de confirmação enviadas pelos demais processos.

No caso dos *rollbacks* simultâneos o processo observador inicia o procedimento de *rollback* tratando o primeiro pedido da fila, ao identificar a linha de recuperação para o primeiro pedido, o processo observador a envia

para os demais processos do sistema juntamente com o novo valor da época do sistema. Porém, este não encerra o procedimento de *rollback*, entrando em um estado de espera para verificar se a linha de recuperação encontrada atende a necessidade de todos os processos. Todo processo, ao receber esta linha de recuperação, deve verificar se a linha de recuperação enviada atende ou não a sua necessidade de *rollback*, caso tenha enviado um pedido de *rollback* ao processo observador. É importante salientar que quando um processo envia um pedido de *rollback* este automaticamente recupera o estado, porém não retoma a simulação, fica aguardando a confirmação do processo observador contendo a linha de recuperação do sistema.

Deste modo, quando o processo recebe uma linha de recuperação do processo observador e identifica que sua necessidade foi atendida, este deve enviar uma mensagem de confirmação, indicando ao processo observador que a linha de recuperação recebida foi aceita e a época foi atualizada. Neste caso, todos os pedidos de *rollback* que possuir uma época inferior a atual serão descartados.

Porém, quando um processo identificar que a linha de recuperação enviada pelo processo observador não atende sua necessidade, este deve enviar uma mensagem avisando ao processo observador que sua necessidade não foi atendida. Este, por sua vez, atualizará o pedido de *rollback* deste processo e o tratará em seguida.

3.5. Considerações Finais

Apesar de possuir uma implementação mais complexa que o protocolo *Time Warp*, o protocolo *Rollback* Solidário possui vantagens que permitem que este ocupe uma posição de destaque entre os protocolos otimistas. Para que todas as vantagens oferecidas pelo protocolo *Rollback* Solidário fossem alcançadas, optou-se pela utilização de um processo observador que tem como função armazenar e identificar os *checkpoints* globais consistentes utilizados na recuperação do sistema. Inicialmente, pode-se ter uma falsa impressão de

que o emprego de um processo observador possa inserir prejuízos na computação da simulação distribuída, dado que este processo possui apenas um comportamento passivo durante a simulação. Porém, sua presença é compensada pela velocidade na identificação de uma linha de recuperação e pelo tempo ganho na realização dos *rollbacks* solidários.

Capítulo 4: *Frameworks* para o Desenvolvimento de Aplicações Distribuídas

Este capítulo tem como principal função apresentar uma breve discussão sobre o estado da arte dos *frameworks*. Fazem parte deste capítulo as seguintes explanações: definição de *framework*, diferenças entre *frameworks* e bibliotecas de classes, as classificações de *frameworks* de acordo com sua utilização, vantagens e desvantagens na utilização e como um *framework* deve ser construído.

De acordo com Johnson e Foote (1988), *frameworks* são conjuntos de classes orientadas a objetos que incorporam soluções para uma família de problemas relacionados. Os *frameworks* são desenvolvidos para prestar um serviço aos desenvolvedores de *softwares*, pois oferecem soluções prontas ou semi-prontas de problemas específicos já solucionados por outros programadores.

O *framework* proposto apresenta soluções inovadoras que não foram encontradas em nenhum dos *frameworks* ou bibliotecas de classes estudadas no desenvolvimento deste trabalho. Este novo *framework* apresenta soluções que possibilitam pesquisadores de outras áreas utilizarem aplicações distribuídas, tornando assim uma ferramenta de grande utilidade.

A programação orientada a objetos surgiu para permitir aos programadores reutilizar partes de soluções de um programa já desenvolvido em outros programas, reduzindo, por conseguinte, o custo da manutenção ou do desenvolvimento de um novo programa. A partir do conceito de reutilização foram criadas várias bibliotecas de classes que oferecem soluções prontas e que permitem ao programador um desenvolvimento mais eficiente e eficaz, considerando que estas soluções já foram testadas e validadas por vários programadores. Porém, com o desenvolvimento de *frameworks* o conceito de reutilização se tornou mais refinado, pois conduz a implementação de ferramentas através de componentes que podem ser utilizados sem que o usuário conheça seu código original. O que diferencia as bibliotecas de classes e os *frameworks* é que, neste último, todas as associações, generalizações, dependências e refinamentos já estão desenvolvidos dentro do componente, enquanto que em uma biblioteca de classes são oferecidos apenas as classes desenvolvidas e todo fluxo de controle deve ser implementado pelo usuário (JOHNSON & FOOTE, 1988).

4.1. Classificações de *Frameworks* de Acordo com sua Aplicação e sua Extensibilidade

Os benefícios e princípios de desenvolvimento subjacentes dos *frameworks* são em sua maioria independentes dos domínios em que se aplicam, porém, tem sido vantajoso classificá-los de acordo com sua utilização. Segundo Fayad e Schmidt (1997) os *frameworks* são classificados em:

- ✓ *Frameworks* para sistemas de infra-estrutura: são empregados no desenvolvimento de ferramentas para sistemas como: sistemas operacionais, redes, construção de interfaces e ferramentas de processamento de linguagem. Estes tipos de *frameworks* tornam o desenvolvimento dos sistemas de infra-estrutura mais simples e portáveis, porém possuem alto nível de generalidade e pouco em aplicabilidade, pois incorporam uma grande variedade de soluções para problemas específicos podendo ser utilizados em mais de um domínio. Estas características fazem com que todo sistema em que

são aplicados necessitem ainda de um grande trabalho de implementação até que o desenvolvimento seja concluído. Estes componentes são conhecidos como *frameworks* horizontais e não são normalmente comercializados, devido ao fato de pertencerem quase na sua totalidade a organizações de *softwares* (SUNKPHOT, 2001).

- ✓ *Frameworks* de integração de *middleware*: são concebidos com o intuito de melhorar a capacidade de modularização e reutilização de programas de infra-estrutura a fim de que os mesmos possam funcionar em ambientes distribuídos. Os componentes destes tipos de *frameworks* são projetados para mascarar a heterogeneidade de redes, *hardware* e até mesmo de sistemas operacionais e linguagem de programação. *Frameworks* de integração de *middleware* estão em ascensão, se tornando rapidamente uma ferramenta de ampla utilização por profissionais da área de desenvolvimento de *softwares* (BAKKEN, 2003).

- ✓ *Frameworks* de Aplicação Empresarial: diferentemente dos *frameworks* para sistemas de infra-estrutura e de integração de *middleware*, os *frameworks* de aplicação empresarial possuem funcionalidade vertical, isto é, apresentam alto nível de aplicação e pouca generalização. *Frameworks* verticais são desenvolvidos para uma quantidade específica de domínios com problemas exclusivos (SUNKPHOT, 2001). Porém, são amplamente utilizados por empresas de atividades comerciais, como: empresas de telecomunicações, de manufaturas, financeiras e de sistemas de alarme. Em relação aos outros tipos de *frameworks* estes não são desenvolvidos para darem suportes internos a sistemas, mas sim para apoiar o desenvolvimento de aplicações do usuário e do produto final diretamente. E, por estes motivos, são caros na sua aquisição e no seu desenvolvimento. No entanto, estes *frameworks* geralmente fornecem um retorno substancial para o investimento.

Outra maneira de se classificar um *framework* é de acordo com sua extensibilidade, podendo ser separado em *white box framework* e *black box framework*.

Nos *white box frameworks*, também conhecidos como *architeture-driven frameworks*, é possível criar novas classes através de uma simples instanciação. Desta forma, novas classes podem ser introduzidas no *framework* através de herança e composição. Porém, para que se possam inserir novas classes neste tipo de *framework*, o programador em questão deve conhecer muito bem a ferramenta a fim de complementá-la sem prejudicar o desempenho do *framework* e seu nível de reutilização (MARKEWICZ & LUCENA, 2001).

Já os *black box frameworks*, também conhecidos como *data-driven frameworks*, só podem ser estendidos através de objetos de composição empregando os componentes de interface conforme implementados no *framework* (FAYAD & SCHMIDT, 1997). Assim, usuários de *black box frameworks* devem conhecer muito bem a interface de conexão dos objetos que preenchem os pontos de variabilidade do *framework* e não a sua implementação interna. Por isto, geralmente *black box frameworks* são mais fáceis de se utilizar do que os *white box frameworks*. Outro fator importante encontrado em *black box frameworks* é a facilidade em que as adaptações são realizadas dinamicamente, podendo acontecer em tempo de execução (JOHNSON & FOOTE, 1988).

Black box frameworks possuem uma maior dificuldade de projeto do que *white box frameworks*, pois o programador durante o desenvolvimento deve se preocupar com a elaboração da interface de conexão dos componentes, além de ter que levar em consideração as possibilidades genéricas de usos futuros do *framework*. Para que as oportunidades de reutilização de um *framework* sejam limitadas ou comprometidas, basta apenas que as interfaces de conexão do *framework* em questão sejam elaboradas de forma inadequada (JOHNSON & FOOTE, 1988).

4.2. Vantagens e Desvantagens da Utilização de *Frameworks*

A utilização de *frameworks,* com intuito de reduzir esforços e aumentar a qualidade de *softwares,* é um fato real, porém, necessita que uma série de desafios seja superada para que se possa utilizá-los com eficiência e eficácia. O primeiro desafio é escolher qual *framework* é mais apropriado para a linha de pesquisa ou desenvolvimento da instituição ou empresa que irá empregá-lo. Uma escolha imprópria pode comprometer todo o projeto invalidando todos os benefícios e reduzindo os custos esperados na aquisição de um *framework*. O próximo passo é adquirir o conhecimento da ferramenta adotada, a fim de utilizar todos os recursos disponíveis adequadamente.

Vantagens de aplicações utilizando *frameworks* são:

- ✓ *Frameworks* incorporam e preservam conhecimentos, pois são desenvolvidos a partir de soluções de problemas específicos encontradas por vários programadores (ANDERT, 1994).
- ✓ Várias de suas funcionalidades já estão prontas tornando o desenvolvimento de novos *softwares* mais rápido e de menor custo.
- ✓ As soluções são confiáveis, robustas e de alta qualidade. Isto ocorre pelo fato de geralmente terem sido testadas por um grande número de programadores.
- ✓ Conforme aumenta a sua utilização, o *framework* vai adquirindo certa maturidade. Isto ocorre, pois de acordo com a sua utilização erros são descobertos e eliminados; e novas funcionalidades são incrementadas (ANDERT, 1994; ASSIS & SUZANO, 2003).
- ✓ Com o emprego dos *frameworks* um número menor de linhas do programa é escrito, diminuindo a possibilidade de erros simples e comuns (ASSIS, 2003).
- ✓ A manutenção é rápida e fácil, pois para corrigir ou criar alguma funcionalidade, geralmente não há a necessidade de se alterar todo o código do *framework*, apenas as partes relacionadas (ANDERT, 1994).

Apesar destas vantagens existem algumas desvantagens, sendo elas:

- ✓ A utilização de *framework* requer pessoas especializadas. Como os *frameworks* contém muitas soluções é necessário que os desenvolvedores passem por um treinamento ou realizem um estudo aprofundado sobre a ferramenta, com o intuito de conhecer as interfaces de conexão, no caso de *black box frameworks*, ou as classes utilizadas, *white box frameworks*, demandando mais tempo para se iniciar a aplicação do produto (ASSIS, 2003).
- ✓ Em *frameworks* imaturos a depuração de programas pode se tornar complexa se o fabricante do *framework* não disponibilizar os códigos fonte, visto que estes podem conter erros de implementação em seus objetos.
- ✓ Em alguns casos, principalmente em *frameworks white box*, a linguagem de programação de desenvolvimento de *software* ficará restrita a linguagem utilizada pelo *framework* perdendo, em alguns casos, sua portabilidade.

Embora possua desvantagens, a viabilidade da utilização de *frameworks* no desenvolvimento de novos *softwares* está na agilidade e redução de custos nos projetos subseqüentes envolvendo a mesma ferramenta.

4.3. Descrição de Algumas Bibliotecas e *Frameworks* Dedicados a Ambientes Distribuídos

Diversas ferramentas são desenvolvidas para auxiliar na utilização de ambientes distribuídos e várias técnicas empregadas por estas ferramentas foram inseridas no *framework* que será apresentado no próximo capítulo. Com o propósito de contextualizar estas técnicas, nesta seção serão apresentadas algumas bibliotecas de classes e *frameworks* desenvolvidos para ambientes distribuídos.

Como já discutido as bibliotecas de classes se diferenciam dos *frameworks* principalmente por não possuírem interconexão entre elas. Todo fluxo de dados e toda relação entre as classes devem ser desenvolvidas durante o projeto. As classes que serão expostas nesta seção são utilizadas

para serializar ou encapsular objetos e dados, e fornecer uma maior generalização das informações, permitindo que estas possam ser inseridas em *buffers* e transmitidas através de um grande número de ferramentas de comunicação.

A *ClassdescMP* (RUSSEL & MADINA, 2003) foi desenvolvida para ser utilizada por programadores de *C++* utilizando *MPI.* Sua função é facilitar a construção das mensagens que serão trocadas entre os processos lógicos de um sistema distribuído simplesmente. E esta biblioteca não foi implementada para encapsular as chamadas do *MPI*, mas sim, com o propósito de complementá-las. Desta forma, todas as chamadas já existentes na biblioteca *MPI* podem ser realizadas diretamente, permitindo que o programador aumente o desempenho do sistema quando houver oportunidade.

Por muito tempo o *MPI* vem sendo empregado com sucesso em sistemas que necessitam realizar troca de informações, como palavras e números. Entretanto, com o surgimento da programação orientada a objetos, tornou-se necessário enviar e receber objetos. Para isto, o *Classdesc* implementa a utilização de descritores de classes (*class descriptors*), que se assemelha com um compilador, que compila um conteúdo binário reestruturando-o em um objeto novamente. Desta forma, os objetos podem ser serializados e enviados.

OOPS (*Object-Oriented Parallel System*) (SONODA, 2006) é um *framework* desenvolvido para apoiar o desenvolvimento de aplicações paralelas, utilizando programação concorrente, empregando classes concretas e abstratas. Desenvolvido utilizando orientação a objetos, o *OOPS* apresenta uma abstração de alto nível, fazendo com que não haja necessidade do programador se envolver diretamente na implementação do paralelismo da aplicação distribuída. Outro recurso deste *framework* é a possibilidade de inserir outros códigos em sua estrutura de acordo com a necessidade do usuário.

OODFw (Object-Oriented Distributed Framework) é um *framework* que tem como função apoiar a programação por dados distribuídos em aplicações irregulares para alto desempenho (DIACONESCU & CONRADI, 2002). Utilizando objetos seqüenciais e distribuídos este *framework* trabalha com fases seqüenciais e paralelas, seguidas de sincronização, enquanto que a

divisão dos dados e a comunicação entre os processos ficam implícito no modelo. Os dados armazenados em objetos distribuídos são particionados automaticamente entre todos os processos existentes no sistema. Além da própria partição, cada processo apresenta uma cópia dos elementos vizinhos, determinados pelo padrão de acesso aos dados.

OOMPI (Object-Oriented MPI) é uma biblioteca de dados que encapsula as funcionalidade do *MPI* em uma interface orientada a objetos (SQUYRES et al., 1996). Desta forma, o *OOMPI* permite ao usuário utilizar os conceitos do *MPI*, sem a especificação de parâmetros como em *MPI*. Esta biblioteca também representa uma camada fina adicionada entre o programa do usuário e o *MPI*, fornecendo abstrações de classes sem promover perdas de desempenho no sistema de passagem de mensagens.

SAMRAI (Structured Adaptive Mesh Refinement Applications Infrastructure) foi construído utilizando programação orientada a objeto. Este *framework* foi construído para apoiar o desenvolvimento de aplicações paralelas de refinamento de malha adaptativo (*Adaptive Mesh Refinement - AMR*), sem que seja necessário o conhecimento de programação paralela por parte do programador da aplicação. Formado por uma coleção de pacotes, o *SAMRAI* apresenta um conjunto de elementos de *software* empregados para construir aplicações *AMR* (WISSINK et al., 2003; WISSINK et al., 2001).

COOL (*Concurrent Object-Oriented Language*) esta biblioteca de classe utiliza-se de estruturas para concorrência, exclusão mútua e sincronização de tarefas. Sendo uma extensão da linguagem *C*++ foi projetada visando o trabalho com sistemas de memória compartilhada. Deste modo, o programador informa o padrão de referência de dados e o sistema em tempo de execução distribui as tarefas e os objetos de modo que as tarefas estejam próximas dos objetos a que fazem referência em hierarquia de memória (CHANDRA et al., 1994; CHANDRA,1995).

Outras ferramentas foram estudadas durante este trabalho, como os *frameworks*: *Active Expression* (SIMONE, 1997), *Janus* (GERLACH et al., 2001), *KeLP* (*Kernel Lattice Parallelism*) (BADEN et al., 2001), *Overture* (BROWN et al., 1999; CHAND, 2005;) e o *POOMA* (*Parallel Object-Oriented Methods and Applications*) (DONGARRA et al., 2003) porém, foram escolhidas

aquelas que possuíam algumas características semelhantes a proposta deste trabalho.

4.4. Construção de um *Framework*

A construção de um *framework* é constituída por quatro etapas principais: definição do modelo de objetos do domínio, identificação dos *hot-spots*, projeto do *framework* e adaptação do *framework*. Estas ações, de acordo com Pree (1999), são necessárias para se desenvolver um *framework*, porém, antes de se iniciar o processo de desenvolvimento, deve-se realizar uma descrição geral dele levando em consideração: a identificação dos pontos de flexibilização e a utilização de padrões com o intuito de diminuir o número de ciclos de iteração necessários.

Algumas questões devem ser levadas em consideração durante o desenvolvimento do projeto. Estas questões foram levantadas por Fayad, Schmidt e Johnson (1999), para agilizar o processo de validação do projeto. E para identificar como desenvolver aplicações específicas que pertençam ao domínio do *framework*, especificar o grau de variabilidade que era necessário para definir as soluções em que o *framework* poderá ser utilizado e não utilizar diferentes aspectos variáveis ao mesmo tempo, mas empregá-los separadamente.

A definição do modelo de objetos do domínio é uma etapa que consiste em identificar os principais elementos do *framework* e as formas como eles se relacionam. Deve-se realizar uma análise entre modelos de objetos de aplicações similares e identificar as características comuns entre eles, a fim de se obter um modelo único de objetos que possam refletir as propriedades das aplicações do domínio específico. Nesta etapa, deve-se identificar os pontos que deverão permanecer fixos, ou seja, não poderão ser alterados.

A próxima fase consiste na identificação dos pontos de flexibilização ou *hot-spots* que tem por base em levantar as partes do *framework* que irão variar de acordo com o contexto em que serão utilizadas e que necessitarão de adaptações para sua reutilização. Estes pontos devem ser identificados e documentados, dado que são sensíveis a alterações do domínio. Vários pontos devem ser verificados com a intenção de identificar os *hot-spots*, como: os

aspectos que diferem de outra aplicação, qual o grau de flexibilidade será desejado e a possibilidade de alterar seu comportamento em tempo de execução.

Após a conclusão das duas fases anteriores, inicia-se o projeto do *framework* unindo as partes fixas definidas na primeira etapa com as partes flexíveis identificadas na segunda, procurando aplicar padrões de projeto para garantir que o objetivo do projeto seja alcançado mantendo a característica de reutilização do *framework.* Por fim, se realiza a adaptação do *framework* identificando componentes redundantes ou que não serão utilizados, adaptando novos componentes com o intuito de gerar novas funcionalidades ao *framework* e corrigindo eventuais defeitos inseridos durante o processo de desenvolvimento. A melhor forma de identificar os componentes que deverão ser eliminados ou alterados é utilizando o *framework* várias vezes, pois há uma grande dificuldade em analisar o domínio ou descobrir erros no *framework* sem que o mesmo seja utilizado. Para isto, devem-se utilizar vários exemplos originais distintos para validar o *framework*, a fim de torná-lo mais genérico e reutilizável.

4.5. Considerações Finais

As desvantagens referentes à utilização de *framworks*, estão relacionadas aos custos tanto intelectuais, quanto financeiros despendidos na fase de conhecimento e escolha do *framework* apropriado. Estes custos tendem a ser ressarcidos substancialmente pela flexibilidade e alto grau de reutilização do *framework*.

Para pesquisadores de diversas áreas, a possibilidade de utilizar ferramentas computacionais sofisticadas como auxílio pela busca de resultados de seus estudos, pode gerar uma grande economia de tempo na construção de seus trabalhos. O surgimento da programação orientada a objeto, juntamente com a criação dos *frameworks* e bibliotecas de classes permitiu que cientistas de outras áreas, com um algum conhecimento de programação, possam implementar ferramentas computacionais de alta eficiência e complexidade para gerar os resultados de seus trabalhos.

Capítulo 5: O Projeto do *Framework* Proposto

Este trabalho de mestrado apresenta um *framework* para facilitar o desenvolvimento de programas de simulação que utilizem uma infra-estrutura distribuída através de uma máquina paralela ou, principalmente, um sistema distribuído. Nos capítulos anteriores foi realizada uma breve revisão de literatura sobre protocolos otimistas e *frameworks* utilizados no desenvolvimento de aplicações paralelas e/ou distribuídas. Este capítulo apresenta a especificação pertinente e discussões necessárias para que o *framework* em questão possa ser utilizado corretamente com todos seus recursos disponíveis. Esta ferramenta foi desenvolvida utilizando programação orientada a objetos, para que se possam realizar as adaptações necessárias com maior facilidade e flexibilidade e aproveitar toda modularização desta técnica de desenvolvimento.

Um ambiente de simulação distribuída apresenta, basicamente, um programa de simulação, que foi implementado utilizando algum protocolo de sincronização, e ferramentas de comunicação, que permitem aos processos da aplicação executar em uma arquitetura paralela e/ou distribuída. Esta descrição permite estruturar os principais componentes em uma arquitetura de *software*, que é uma estruturação em camadas ou módulos.

De uma forma geral, os princípios aplicados para se chegar às camadas são:

- ✓ Uma camada deve ser criada onde houver necessidade de outro grau de abstração.
- ✓ Cada camada deve executar uma função bem definida.
- ✓ Os limites da camada devem ser escolhidos para reduzir o fluxo de informações transportadas entre as interfaces.
- ✓ A quantidade de camadas deve ser suficientemente grande para que funções distintas não precisem ser desnecessariamente colocadas na mesma camada e suficientemente pequeno para que a arquitetura não se torne difícil de controlar.

Neste contexto, um ambiente de simulação distribuída pode ser estruturado em uma arquitetura conforme ilustra a figura 5.1.

Figura 5.1 - Estrutura de um ambiente de Simulação Distribuída

O primeiro nível, formado pela camada Arquitetura, é responsável em manter as informações atualizadas da arquitetura física onde o programa de simulação será executado.

O segundo nível, formado pela camada de comunicação, é responsável pelas trocas de informações realizadas durante a simulação. Estas trocas são feitas através do envio e do recebimento de mensagens, entre os processos

envolvidos na simulação, pelos canais de comunicação do sistema. Esta camada também deve garantir que toda mensagem enviada por um processo será recebida pelo processo receptor e que a ordem cronológica de envio será respeitada no recebimento.

O terceiro nível é formado pelo protocolo responsável em realizar a interface entre as camadas de comunicação e aplicação. Neste nível se encontram os protocolos apresentados nos capítulos anteriores. Esta camada é responsável em tratar os erros de causa e efeito com a intenção de garantir a consistência dos resultados obtidos pela simulação.

O quarto e último nível realiza a interface com o usuário permitindo entrar com os modelos a serem simulados e realizar a coleta dos dados após a simulação.

Com o modelo em camadas, apresentado na figura 5.2, o *framework* alcança um alto nível de flexibilidade, permitindo combinar diversas estruturas de forma simples e eficiente. É possível, por exemplo, utilizar uma implementação do protocolo *Time Warp* construído com a biblioteca de troca de mensagens *PVM (Parallel Virtual Machine)* (BEGUELIN, 1994) ou uma implementação do protocolo *Rollback* Solidário utilizando a biblioteca *MPI (Message Passing Interface)* (WALKER, 1994; MCBRYAN, 1994).

A seguir será apresentada a modelagem do *framework* em diagramas de classes e de seqüência da *UML (Unified Modeling Language)* (BOOCH et al., 2000).

5.1. A Modelagem do Sistema

Como o principal objetivo deste trabalho é proporcionar soluções para usuários que desejam realizar simulações de modelos discretos em ambientes distribuídos, este capítulo apresenta, através dos diagramas de classe e seqüência, os recursos disponíveis para a utilização de protocolos que garantem a integridade dos resultados da simulação, assim como a escolha de tais protocolos, em particular os otimistas: *Time Warp* e *Rollback* Solidário.

Outro recurso oferecido é a possibilidade de se implementar a migração de processos e a troca dinâmica entre os protocolos aumentando, assim, a eficiência dos programas de simulação.

Diagramas de classes são representações estáticas e, por oferecer uma visão lógica do sistema, permitem descrever como as funcionalidades foram ou serão desenvolvidas. Sua estrutura é válida em qualquer ponto da execução do sistema, se tornando uma ferramenta de modelagem muito útil na sua compreensão.

Através da descrição das classes do modelo desenvolvido, todos os elementos fixos e flexíveis do *framework*, além de suas relações, serão apresentados e discutidos, com o propósito de facilitar a compreensão dos recursos disponíveis para o desenvolvimento de aplicações específicas.

A classe principal deste *framework* é a classe `Ambiente`. Esta é uma classe contêiner que define o ambiente principal que hospedará os principais objetos da aplicação do usuário, conforme pode ser visto na figura 5.2. A classe `Ambiente` tem a função principal de abstrair a hierarquia apresentada na figura 5.1, ou seja, separar a arquitetura física onde ocorrerá a simulação, o ambiente de passagem de mensagens que será utilizado, o protocolo de simulação distribuída e o modelo que será simulado.

A classe `Ambiente` é composta de quatro objetos principais: um objeto da classe `Protocolo`, um objeto da classe `Comunicacao`, um objeto da classe `Arquitetura` e outro objeto da classe `EstadoGeral`. As classes `Protocolo`, `Comunicacao` e `EstadoGeral` são classes abstratas para proporcionar maior flexibilidade para o usuário do *framework*. A classe `EstadoGeral` tem por finalidade armazenar informações gerais do comportamento da simulação, ou seja, manter as informações que se deseja analisar. Apesar de não estar explicitamente indicado no diagrama, esta classe possui uma relação lógica com a classe `Modelo`, pois o usuário deve especificar, na classe descendente concreta, as informações que deseja analisar. As classes `Arquitetura` e `Protocolo` serão descritas nas seções seguintes.

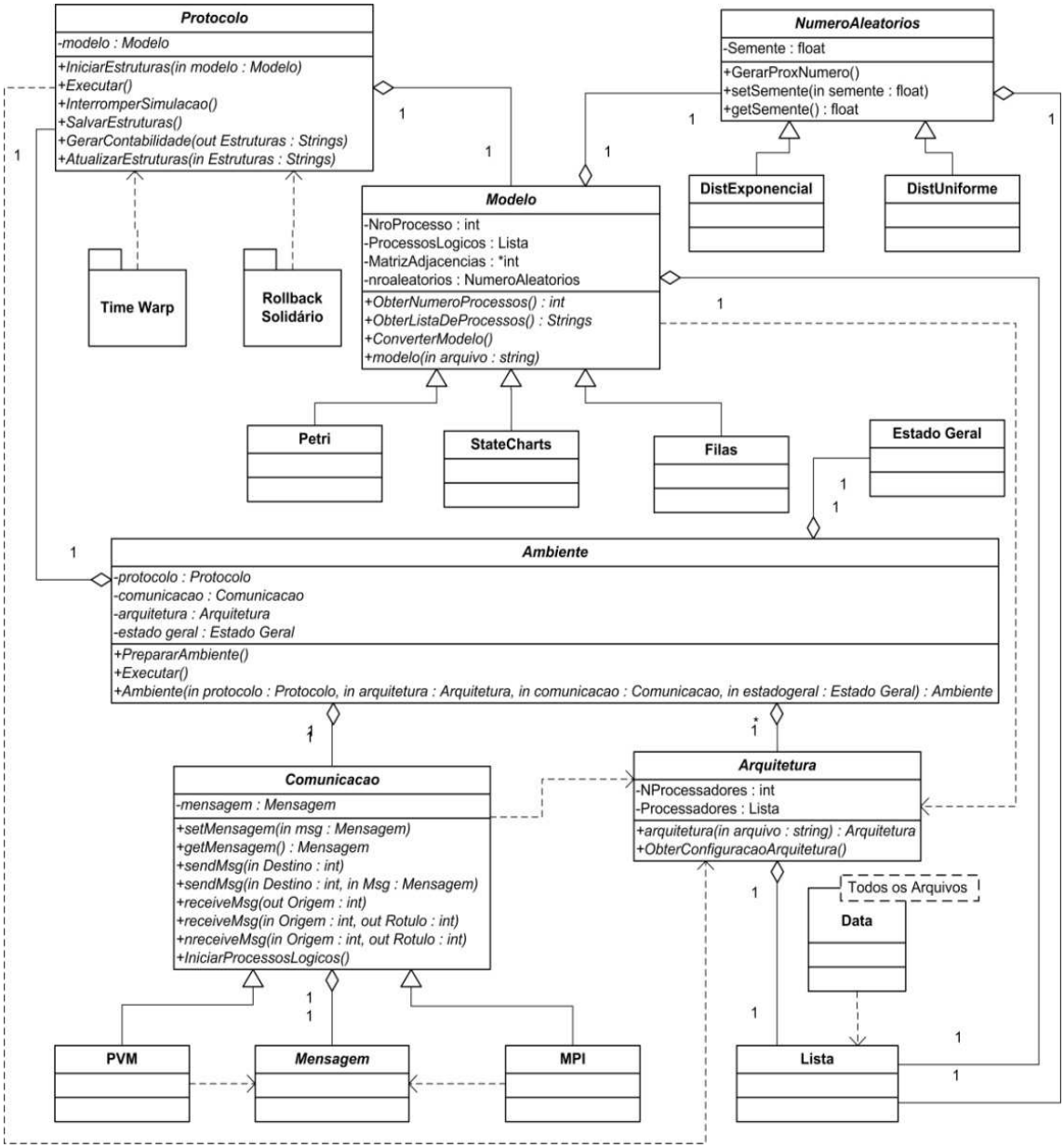

Figura 5.2 - Diagrama de classes do *framework* proposto

Além destes atributos, a classe `Ambiente` possui dois métodos principais: `PrepararAmbiente()` e `Executar()`. O primeiro método invoca os métodos de inicialização dos objetos das classes `Arquitetura` e `Protocolo` que, por sua vez, também envia a mensagem de preparação para o objeto da classe `Modelo`, escalonando os respectivos processos lógicos aos processadores, conforme estrutura definida na classe `Arquitetura`. É importante destacar que o construtor da classe recebe os objetos concretos das quatro classes que a compõe.

5.1.1. A Classe Arquitetura

As arquiteturas paralelas podem ser definidas como “um conjunto de elementos de processamento que podem se comunicar e cooperar para resolver grandes problemas rapidamente” (ALMASI & GOTTLIEB, 1994). Essa definição cria uma abstração sobre o conceito de arquiteturas paralelas, permitindo que sejam elaboradas questões relativas aos elementos de processamento, aos mecanismos de comunicação, a cooperação entre esses elementos de processamento e sobre o que significa resolver grandes problemas rapidamente.

Toda a descrição lógica da arquitetura física do sistema fica sobre a responsabilidade da classe `Arquitetura`. São informações como o número de estações da rede (não necessariamente o número de processos lógicos da simulação) e as características de cada um destes processadores. Estas informações são armazenadas em uma lista onde cada nó contém dados sobre: os recursos de processamento de cada estação, a quantidade de memória e outras informações necessárias para as tomadas de decisão a respeito do mapeamento dos processos envolvidos.

Destaca-se que as informações da arquitetura são passadas para o objeto através dos dados armazenados em um arquivo texto previamente configurado. A figura 5.3 ilustra um exemplo de arquivo que define a estrutura do sistema distribuído existente no laboratório do grupo de pesquisa *GPESC* (*Grupo de Pesquisa em Engenharia de Sistemas e de Computação*) da Universidade Federal de Itajubá. Este arquivo é dividido em seções. Cada seção tem uma função específica. No exemplo apresentado há apenas duas seções: Elementos e Estações. A primeira seção possui um campo Quantidade que define a quantidade total de nós da arquitetura. A segunda seção define um conjunto de informações para cada tipo de *hardware* disponível, assim, como pode ser observado, existem 32 computadores *Intel Quad Core* e 2 coputadores *Intel Xeon Dual Zion*.

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

[Elementos]
Quantidade: 34

[Estações]
Quantidade: 32
Arquitetura: Intel Quad Core Q6600
Processador: 2,40GHz
Memória: 2GB
Cache: 8MB
Disco: 250GB

Quantidade: 2
Arquitetura: Intel Xeon Dual Zion 5410
Processador: 2,33GHz
Memória: 16GB
Cache: 12MB
Disco: 4,5TB
```

Figura 5.3: Arquivo de Definição da Estrutura do Sistema Distribuído

5.1.2. A Classe Modelo e Classe NúmerosAleatórios

A classe `Modelo` é abstrata para permitir que modelos elaborados em ferramentas de modelagem diferentes como, por exemplo: Redes de Filas, *State Charts* e Redes de Petri, possam ser utilizadas com as classes do *framework*. Para isto, será necessário sobrescrever o método abstrato `ConverteModelo()`, que tem a função de atualizar os atributos da classe, nas classes concretas descendentes.

Informações a respeito do modelo simulado serão tratadas na classe `Modelo` como um grafo dirigido. Há uma lista contendo informações sobre cada processo lógico como, por exemplo, a probabilidade de ser criado um novo evento a cada iteração da simulação, as probabilidades de que sejam criados eventos para outros processos lógicos, durante o tratamento de um evento, e a probabilidade de que um evento termine sem gerar novas atividades.

Semelhante à classe `Arquitetura`, a classe `Modelo` recebe as informações pertinentes ao modelo que o usuário pretende simular através de um arquivo de configuração. Isso permite maior flexibilidade e ainda possibilita a simulação de diversos modelos sem a necessidade de escrever um programa para cada um deles. Se a simulação vai ocorrer no mesmo ambiente, utilizando o mesmo protocolo e com a mesma biblioteca de troca de mensagens, basta alterar o arquivo com as informações do modelo e re-executar o programa.

A figura 5.4 ilustra um exemplo de arquivo de configuração e a figura 5.5 apresenta o grafo que representa o modelo a ser simulado.

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

[Processos]
Quantidade: 3

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

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

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

Figura 5.4: Arquivo de Configuração

O modelo, apresentado nas figuras 5.4 e 5.5, possui três processos lógicos. Todos utilizam distribuição exponencial para a distribuição de chegada.

Esta informação é utilizada durante a geração de números aleatórios. O primeiro processo possui uma taxa de 50% de probabilidade de chegada a cada iteração e, durante o processamento de cada evento, existe 30% de chance de o evento encerrar sem gerar novos eventos. Este processo não gera eventos para si, mas tem 20% de chance de gerar um evento para o ***processo 2*** e 50% de probabilidade de gerar um evento para o ***processo 3***. O raciocínio é idêntico para os processos ***2*** e ***3***.

A simulação de sistemas requer o uso de seqüências de valores de determinadas variáveis aleatórias. Segundo Perin (1995), pode-se considerar que há três modos de obter tais seqüências:

- ✓ O uso de seqüências provenientes de observações efetuadas previamente;
- ✓ O uso de seqüências geradas aleatoriamente a partir de distribuições empíricas construídas com observações efetuadas previamente;
- ✓ O uso de seqüências geradas aleatoriamente a partir de distribuições clássicas cujos parâmetros foram estimados de acordo com observações efetuadas previamente.

Desta forma, como existem diversos métodos para geração de números aleatórios, o objetivo da classe `NumerosAleatorios` é permitir que sejam implementados os algoritmos das distribuições de probabilidade necessárias à simulação dos modelos dos usuários. É, desta forma, também uma classe abstrata devido ao o método `GerarProxNumero()`. Basicamente, a classe mantém um valor inicial (semente) e, partir deste valor, este método deverá gerar a nova semente.

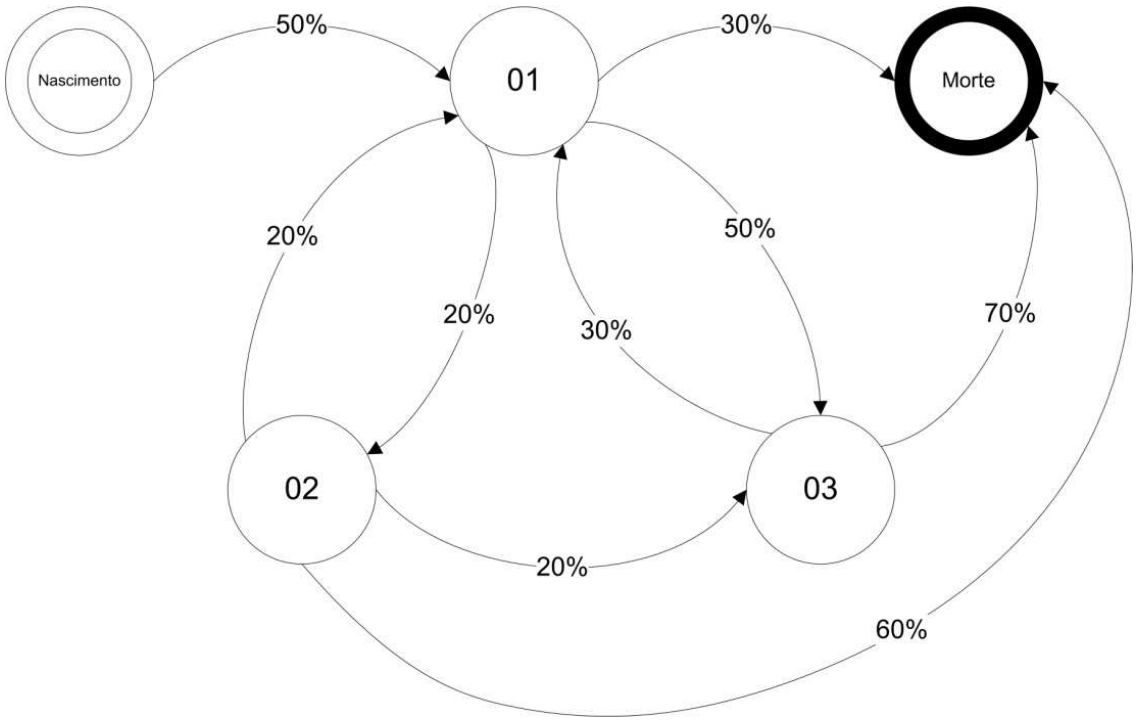

Figura 5.5: Grafo Representativo do Modelo Simulado

5.1.3. As Classes `Comunicacao` e `Mensagem`

A classe `Comunicacao` é uma classe abstrata, pois representa as funções de comunicação existentes nos ambientes de troca de mensagens do sistema. De tal modo, a partir desta classe, podem-se criar classes concretas que implementem as primitivas de comunicação existentes nos ambientes de interesse do usuário.

Esta classe não possui atributos, contém apenas as assinaturas dos métodos abstratos que deverão ser sobrescritos nas classes concretas descendentes. Essas classes descendentes possuirão, basicamente, dois atributos: um vetor com os identificadores dos processos, que farão parte do ambiente de comunicação, e um objeto com a estrutura das mensagens que trafegarão na rede. A implementação destes métodos dependerá da biblioteca de troca de mensagens utilizada. Como exemplo, a listagem 5.1 apresenta uma implementação, em *C++*, simplificada da classe concreta *PVM*.

Listagem 5.1 - Implementação da classe concreta *PVM*

```
class Comunicacao{
private:
  int *PIds;
  int NProc;
  Mensagem *Msg;
public:
  Comunicacao(int, int *);
  int getId(int);
  int getNProc();
  void setNProc(int);
  void setMensagem(Mensagem *);
  Mensagem* getMensagem();
  void sendMsg(int);
  void sendMsg(int, Mensagem *);
  void receiveMsg(int = -1, int = -1);
  int nreceiveMsg(int = -1, int = -1);
};
Comunicacao::Comunicacao(int nproc, int *pids){
  NProc = nproc;
  PIds = new int[nproc];
  for (int i=0; i<NProc; i++)
    PIds[i] = pids[i];
}
int Comunicacao::getId(int indice){
  return PIds[indice];
}
int Comunicacao::getNProc(){
  return NProc;
}
void Comunicacao::setMensagem(Message *M){
  Msg = M;
}
Mensagem* Comunicacao::getMensagem(){
  return Msg;
}
void Comunicacao::sendMsg(int Target){
   pvm_initsend(PvmDataRaw);
   pvm_pkint(Msg->getContent(), Msg->getSize(), 1);
   pvm_send(Target, Msg->getMsgKind());
}
void Comunicacao::sendMsg(int Target, Message *M){
  setMensagem(M);
  sendMsg(Target);
}
void Comunicacao::receiveMsg(int Target, int Label){
  int Aux[SIZE];
```

```
  pvm_recv(Target, Label);
  pvm_upkint(Aux, SIZE, 1);
  Mensagem *M = new Mensagem(Aux, SIZE);
  setMensagem(M);
}
int Comunicacao::nreceiveMsg(int Target, int Label)
{
  int Aux[SIZE];
  if(pvm_nrecv(Target, Label)){
    pvm_upkint(Aux, SIZE, 1);
    Mensagem *M = new MensagemPVM(Aux, SIZE);
    setMensagem(M);
    return 1;
  } else
    return 0;
}
```

A classe abstrata `Mensagem` permite ao usuário do *framework* adaptar a estrutura desta classe para os dados específicos que ele deseja manipular durante a `Comunicacao` dos processos do seu programa de simulação. Além disso, podem-se construir classes descendentes que explorem as características próprias da biblioteca de troca de mensagens utilizada. Desta forma, o usuário, ao construir uma classe descendente de comunicação para uma determinada biblioteca, também construirá uma classe para ser a estrutura das mensagens que serão utilizadas.

5.1.4. A Classe `Protocolo`

A classe `Protocolo` possui mecanismos que simplificam a utilização dos protocolos de sincronização. Esta classe está associada aos componentes que compõem as estruturas de cada implementação de um protocolo. É também uma classe abstrata, pois para cada tipo de protocolo, será necessário criar uma classe descendente concreta. Esta classe descendente é, na verdade, o ponto de ligação do *framework* com o pacote que corresponderá às classes que, efetivamente, descreverão o funcionamento do protocolo em questão. Isto pode ser visto através das classes `TW` e `RS` e respectivas associações com os pacotes “`Time Warp`” e “`Rollback Solidário`”.

O método `IniciarEstruturas()` prepara as estruturas do protocolo para a simulação. Este método, associado ao protocolo *Time Warp*, por

exemplo, inicia as listas de eventos futuros, listas de mensagens e de anti-mensagens, além de realizar o salvamento do estado inicial de cada processo lógico.

O método `InterromperSimulacao()` é um método que pode ser invocado externamente caso o usuário deseje verificar como está o andamento da simulação, ou seja, avaliar os dados parciais produzidos até um determinado instante. Além disso, este método pode ser utilizado para implementar um mecanismo de troca dinâmica de protocolos. Neste caso, porém, seria necessário invocar o método `SalvarEstruturas()` para armazenar as respectivas informações. A partir deste ponto, a simulação poderia ser reiniciada com outro protocolo.

No diagrama principal do *framework* (figura 5.2), há dois pacotes relacionados com a classe `Protocolo`: "`Time Warp`" e "`Rollbak Solidario`". Em *UML*, os agrupamentos que organizam um modelo são chamados de pacotes. Um pacote é um mecanismo de propósito geral empregado para organizar os elementos em modelos, de maneira que seja mais fácil compreendê-los. São utilizados também para permitir o controle de acesso a seus conteúdos, de modo que seja possível controlar as costuras existentes na arquitetura do sistema (BOOCH et al., 2000)

A classe `Protocolo` é uma classe que realiza o gerenciamento dos pacotes correspondentes às implementações dos protocolos para simulação distribuída. Essa classe também funciona como interface entre estes pacotes e as demais classes do *framework*. Qualquer protocolo que possa ser implementado mantendo as regras de comunicação com a classe `Protocolo` pode se tornar um protocolo disponível no *framework* para utilização dos usuários em geral, até mesmo protocolos conservativos, como é o caso do protocolo *CMB*. A classe `Protocolo` é uma classe abstrata devido aos métodos `GerarContabilidade()` e `SalvarEstruturas()`.O método `SalvarEstruturas()` tem a função de armazenar o estado das variáveis internas de gerenciamento do protocolo. Desta forma, quando a simulação necessitar ser retomada, estes dados serão utilizados para reiniciar a simulação do ponto onde ocorreu a interrupção.

O método `GerarContabilidade()` retorna uma lista de *Strings* contendo informações a respeito do protocolo. No protocolo *Time Warp* exemplos destas de informações são: a quantidade de *rollbacks* realizados, o tamanho médio da fila de eventos futuros e o tamanho médio da fila de anti-mensagens, além dos estados atuais destas estruturas; os números de eventos realmente realizados (que não serão mais afetados por um *rollback*) e a quantidade de eventos desfeitos devido aos procedimentos de *rollback*. No *Rollback* Solidário, além destas informações, a lista acrescenta o número de *checkpoints* básicos e o número de *checkpoints* forçados realizados pelo método semi-síncrono adotado. Estas informações são essenciais para a implementação de um procedimento de troca dinâmica de protocolos. Também é possível comparar o desempenho do protocolo corrente em tempo de execução.

O método `AtualizarEstruturas()` é um método que permite iniciar as estruturas do protocolo com dados previamente configurados que são passados através de uma lista de *strings*. Esta lista tem a mesma estrutura daquela retornada pelo método `GerarContabilidade()`. Através deste método é possível iniciar um protocolo com dados de uma simulação que foi parcialmente realizada anteriormente. Assim, este método será utilizado em um mecanismo de troca de protocolos, ou mesmo, para continuar uma simulação que foi interrompida anteriormente, permitindo a criação de mecanismos de persistência da simulação.

Estes métodos devem ser tratados nas classes concretas descendentes da classe `Protocolo`, ou seja, devem ser sobrescritos nas classes concretas para implementar as funcionalidade descritas.

Para que fique claro como devem ser tratados as implementações de cada protocolo, este trabalho inclui os dois protocolos otimistas discutidos no capítulo dois e três: *Time Warp* e *Rollback* Solidário. As classes que compõem o protocolo *Rollback* Solidário seguem a estrutura apresentada por Moreira (2005), excetuando-se a existência de uma classe denominada `Ambiente`, proposta no trabalho de Moreira (2005), cujo objetivo está totalmente inserido no contexto das outras classes presentes neste *framework*. Além disso, no

referido trabalho, a classe Processo é uma classe abstrata, pois a implementação do protocolo pode se dar com a utilização de um processo observador ou sem a existência deste processo. Quando não há um processo observador, cada processo se torna responsável em gerenciar o procedimento de *rollback* quando recebe uma mensagem *straggler*. Entretanto, mesmo que seja desenvolvida uma versão do protocolo com esta característica, a existência do processo observador passa a ser necessária, não somente para coordenar o procedimento de *rollback*, mas também para gerenciar procedimentos de troca de protocolo ou de migração de processos. Por estes motivos, a classe Observador passa a ser uma classe abstrata. A classe concreta Observador_SR tem as características da classe Observador apresentada no trabalho de Moreira (2005). Ou seja, possui a estrutura necessária para implementar o processo observador e os métodos que se destacam são: ExtrairLinhaDeRecuperacao(), ObterLinhaDeRecuperacao() e ProcederRollback() que implementam as atividades que foram descritas no capítulo três. Ambas as classes, Processo e Observador, possuem o método ObterMensagem() para facilitar a interação com o objeto da classe Receptor. Para atender as novas funcionalidades, foram acrescentados, na classe Observador, os métodos abstratos EscalonarProcessos(), IniciarTrocaDeProtocolo() e FinalizarTrocaDeProtocolo(). O método EscalonarProcesso() permitirá redistribuir os processos se for implementado um mecanismo de migração. Os outros dois métodos serão utilizados em um mecanismo de troca dinâmica de protocolos, sendo que o método IniciarTrocaDeProtocolo() permitirá paralisar as estruturas no protocolo corrente e o método FinalizarTrocaDeProtocolo()iniciará as estruturas no protocolo destino da troca.

O diagrama de classes do protocolo *Rollback* Solidário, representado pela figura 5.6, apresenta as principais classes de sua estrutura. A classe Processo é a base do modelo e trata-se de uma classe abstrata, uma vez que a declaração do método executar() é apenas um compromisso de que as classes descendentes irão implementá-lo. A classe concreta ProcessoComObservador é descendente direta da classe Processo e implementa o método executar().

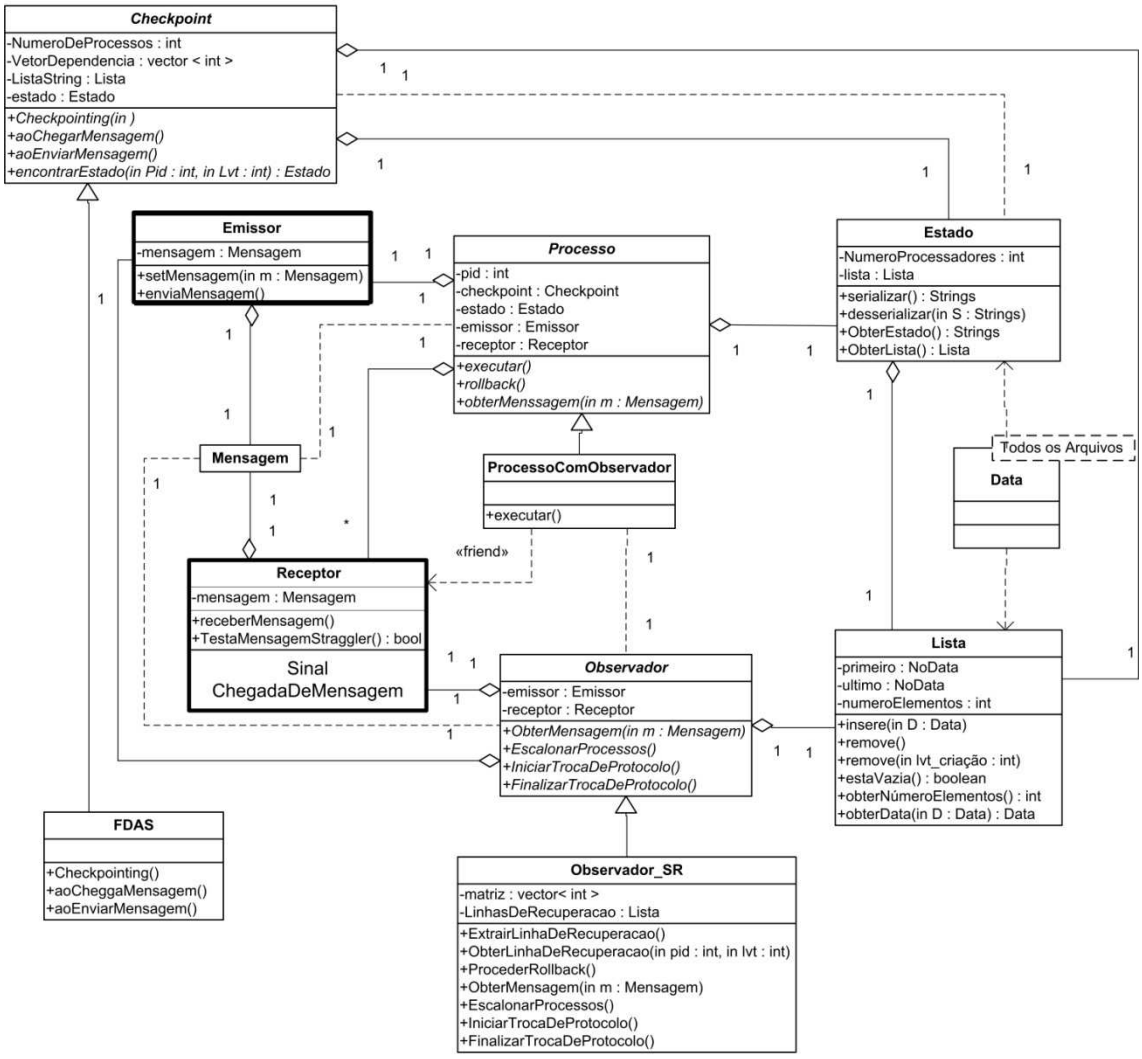

Figura 5.6: Diagrama de Classes do Protocolo *Rollback* Solidário

Além da classe `Processo` e da classe `Observador`, a classe `Checkpoint` também é uma classe abstrata. Isto se deve ao fato de que diferentes algoritmos semi-síncronos podem ser utilizados para se obter *checkpointing*. Para cada algoritmo, deve-se criar uma nova classe descendente de `Checkpoint` e implementar o algoritmo através dos métodos `Checkpointing()`, `aoChegarMensagem()`e `aoEnviarMensagem()` que representam, respectivamente, o método para salvar os estados dos processos durante o procedimento de *checkpoint* e os métodos para atualizar o vetor de dependências e realizar os *checkpoints* forçados durante a troca de mensagens entre os processos lógicos do programa de simulação.

A classe Estado representa os atributos de cada processo da simulação. A separação destes atributos da classe Processo ocorre devido à necessidade de armazenar os estados durante o procedimento de *checkpointing*. Além disso, esta classe pode ser remodelada de acordo com a necessidade do usuário sem grandes impactos no sistema. A principal característica desta classe é a existência dos métodos serializar() e desserializar() que são responsáveis pela conversão dos atributos dos objetos da classe em uma cadeia de caracteres e vice-versa. Isto facilita o procedimento de *checkpointing* e a restauração dos respectivos estados durante um *rollback*. O método ObterEstados devolve os dados dos atributos do objeto através de uma lista de *strings*.

As classes Lista e Data são classes que implementam as estruturas de dados necessárias para o gerenciamento das listas e filas de eventos futuros da simulação. Várias classes do modelo utilizam a estrutura de dados lista encadeada, como por exemplo, as classes Checkpoint, com o atributo ListaStrings, e Observador, com o atributo LinhasDeRecuperacao. Por conseguinte, a classe Data é uma classe-template, que é um elemento parametrizado. O resultado da instanciação de uma classe-template é uma classe concreta que pode ser empregada da mesma forma que outras classes comuns. Os objetos da classe Data deverão ser instanciados de acordo com o tipo de dados que a classe Lista irá manipular.

O diagrama de classes apresenta duas classes ativas, isto é, que representam comportamentos concorrentes do mundo real e são utilizadas para criar um modelo que use os recursos do sistema o mais eficiente possível. As classes Emissor e Receptor são classes ativas e são responsáveis pela comunicação com as rotinas do nível 1 da arquitetura em camadas do ambiente de simulação, ou seja, os objetos desta classe são responsáveis pelo tratamento das mensagens que são enviadas e recebidas pelos processos lógicos do sistema. A existência destas classes especiais permite a implementação de linhas de controle (*threads*) independentes para o tratamento das mensagens do sistema, permitindo um desempenho superior da simulação. Além disso, a classe Receptor é modelada como classe amiga *(<<friend>>)* da classe ProcessoComObservador para facilitar a comparação dos tempos lógicos das mensagens com o estado do objeto.

Na modelagem, a comunicação entre objetos ativos é descrita utilizando eventos, sinais e mensagens. Um evento é algo que ocorre no sistema ou no ambiente, como a chegada de uma mensagem na rede de comunicação. Os sinais são um caso especial de eventos nomeados que podem ser suspensos. Na classe `Receptor`, o sinal `ChegadaDeMensagem` avisa a chegada de uma mensagem na rede de comunicação.

As duas classes ativas se relacionam com a classe `Mensagem` que contém a estrutura da mensagem de comunicação. Os atributos desta classe foram definidos na seção anterior.

O protocolo *Time Warp* apresenta uma estrutura semelhante à do protocolo *Rollback* Solidário, conforme pode ser visto no diagrama de classes da figura 5.7. Em especial, há a classe abstrata `Checkpoint` que não possui as mesmas funções da correspondente na implementação do protocolo *Rollback* Solidário, pois não utiliza um algoritmo semi-síncrono para a obtenção dos estados. Desta forma, as classes concretas descendentes são implementações dos mecanismos de gerenciamento de memória como, neste caso, o *SSS (Sparse State Saving)*. Outra diferença deste diagrama encontra-se na classe `Observador`. Como discutido anteriormente, no diagrama do protocolo *Rollback* Solidário, esta classe tem a função de encontrar as linhas de recuperação e controlar o procedimento de *rollback*. No protocolo *Time Warp*, ou em outros protocolos que forem adicionados ao *framework*, esta classe terá a função de permitir a implementação de mecanismos para a troca dinâmica de protocolos ou a migração de processos. Desta forma, são mantidos apenas os métodos abstratos `EscalonarProcessos()`, `IniciarTrocaDeProtocolo()` e `FinalizarTrocaDeProtocolo()`.

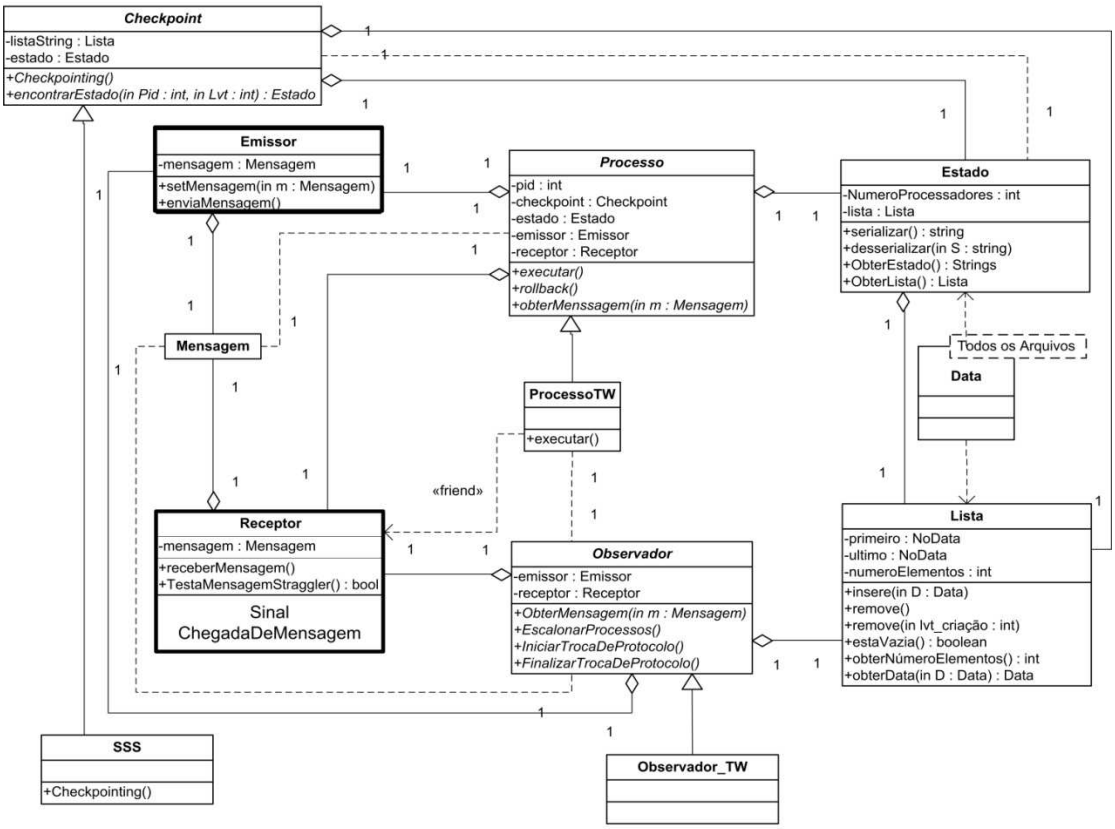

Figura 5.7: Diagrama de Classes do Protocolo *Time Warp*

5.2. Diagramas de Seqüência

Esta seção discute a utilização do *framework* por parte do usuário para o desenvolvimento de aplicações de simulação. Para isso, serão apresentados os diagramas de seqüência que ilustram o comportamento e a inter-relação das classes do sistema.

A listagem 5.2 apresenta um exemplo de código que ilustra a utilização das classes do *framework*. Trata-se de uma função principal em *C/C++.* Inicialmente, é instanciado um objeto da classe `Modelo` denominado `ModeloParaSimulacao`. Como foi discutido na seção 5.1.3, o objeto `Modelo` mantém um grafo representando recursos e as interações entre eles, além da função de distribuição e a probabilidade de comunicação de cada elemento do modelo. Em seguida, o protocolo *Time Warp* é escolhido para sincronizar os processos através da instanciação do objeto `TimeWarpProtocol`. A biblioteca

de troca de mensagens escolhida será o *PVM (Parallel Virtual Machine)* e a estrutura física da rede de computadores utilizada é descrita através da especificação contida no arquivo “`Arquitetura.dat`”. Este arquivo é utilizado pela classe `Arquitetura`. Após a declaração destes objetos, a classe `Ambiente` pode ser instanciada. Isto ocorre com a chamada do construtor que recebe como parâmetros os objetos `SistemaDistribuido`, `TrocaMensagem` e `TimeWarpProtocol`.

Com os objetos preparados, a simulação pode ser iniciada. Para isso, é preciso invocar o método `PrepararAmbiente()` da classe `Ambiente.` Este método ira invocar os métodos necessários para iniciar os processos lógicos da simulação e mapear estes processos nas estações apropriadas, de acordo com a descrição contida na arquitetura e com os critérios de escalonamento utilizados pelo protocolo. Após este passo, a simulação inicia com a chamada ao método `Executar()` da classe `Ambiente`.

Listagem 5.2: Exemplo de utilização do *framework* para a criação de um programa de simulação.

```
#include "framework.h"
int main()
{
  Modelo *ModeloParaSimulacao = new Modelo ("Modelo.dat");
  Protocolo *TimeWarpProtocol =
    new TimeWarp(ModeloParaSimulacao);

  Comunicação *TrocaMensagem = new PVM();

  Arquitetura *SistemaDistribuido =
    new Arquitetura ("Arquitetura.dat");

  Ambiente *AmbienteSimulacao =
    new Ambiente(SistemaDistribuido, TrocaMensagem,
    TimeWarpProtocol);

  AmbienteSimulacao->PrepararAmbiente();

  AmbienteSimulacao->Executar();

  TimeWarpProtocol->GerarContabilidade("Saida.dat");

  return 0;
}
```

5.2.1. O Método PrepararAmbiente()

Para tornar mais claro a compreensão do método PrepararAmbiente(), a figura 5.8 apresenta o respectivo diagrama de seqüência ilustrando as ações que ocorrem quando este método é invocado.

Quando o método PrepararAmbiente(), da classe Ambiente, é invocado, a primeira ação realizada é o envio da mensagem IniciarEstruturas() ao objeto da classe Protocolo. Esta mensagem faz com que o objeto da classe Protocolo obtenha a quantidade de processos lógicos do modelo a ser simulado e, em seguida, a lista contendo as respectivas informações de cada processo. Com estas informações, o método IniciarEstruturas() do objeto da classe Protocolo é encerrado, retornando a configuração do modelo a ser simulado. O método PrepararAmbiente() envia a mensagem ObterConfiguracaoArquitetura() para o objeto Arquitetura que retorna com a estrutura disponível para a execução do programa de simulação. Finalmente, o método IniciarProcessosLogicos(), da classe Comunicacao, é invocado. Este método irá iniciar a máquina virtual, que no exemplo da listagem 5.2 corresponde à biblioteca de troca de mensagens *PVM*.

5.2.2 O Método Executar()

Uma vez que o ambiente está preparado, a simulação pode ser iniciada. Basicamente, o método Executar(), da classe Ambiente, invocará o método Executar() da classe Protocolo. Este último irá iniciar cada processo lógico da simulação e aguardar que cada um destes processos execute o número de iterações definidas no modelo. Quando a execução de cada processo lógico terminar, o método Executar() encerra sua chamada atualizando as variáveis que contabilizam os dados da simulação. Desta forma, o método GerarContabilidade() pode ser invocado para que os resultados da simulação sejam apresentados e utilizados pelo usuário da simulação.

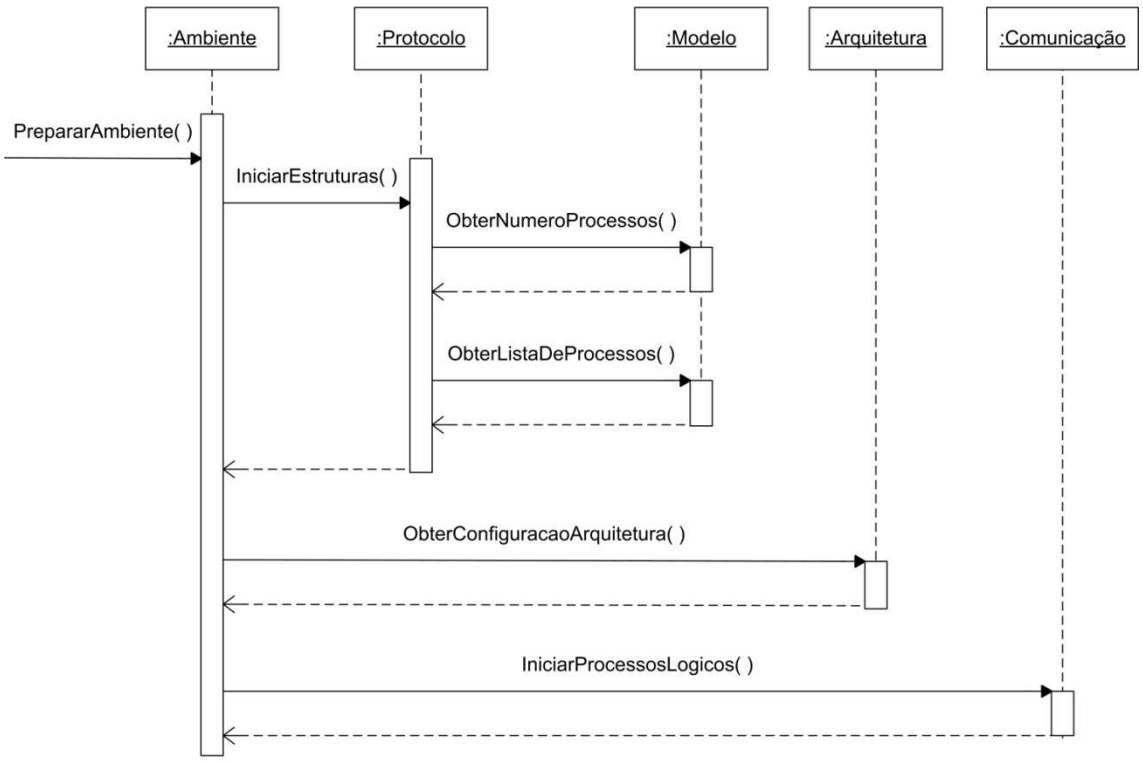

Figura 5.8: Diagrama de Seqüência ilustrando a preparação do ambiente para simulação

5.3. O Comportamento dos protocolos *Time Warp* e *Rollback* Solidário

Os capítulos dois e três apresentaram o funcionamento dos protocolos *Time Warp* e *Rollback* Solidário. Nesta seção, serão apresentados os diagramas de seqüência descrevendo como os métodos das classes das figuras 5.6 e 5.7 são invocados durante a execução da simulação.

5.3.1. Diagramas de Seqüência do protocolo *Time Warp*

O diagrama de seqüência apresentado na figura 5.9 apresenta o comportamento do protocolo *Time Warp* durante um salvamento de estado e durante as trocas de mensagens entre os processos lógicos da simulação. O processo lógico do sistema está representado pelo objeto `Processo`, já o mecanismo de salvamento de estado está implementado no objeto

`Checkpoint`. O mecanismo adotado para esta implementação foi o *Sparse State Save* que tem como característica salvar estados em intervalos específicos e, de certa forma, apresenta uma estrutura semelhante a do protocolo *Rollback* Solidário. O procedimento de salvamento inicia pela chamada ao método `Checkpointing()`. Após esta fase o estado passa por uma serialização, através do método `serializar`() do objeto `Estado`, para que seu conteúdo seja transformado em uma lista de *strings*. Esta estrutura é mantida no objeto `Lista`.

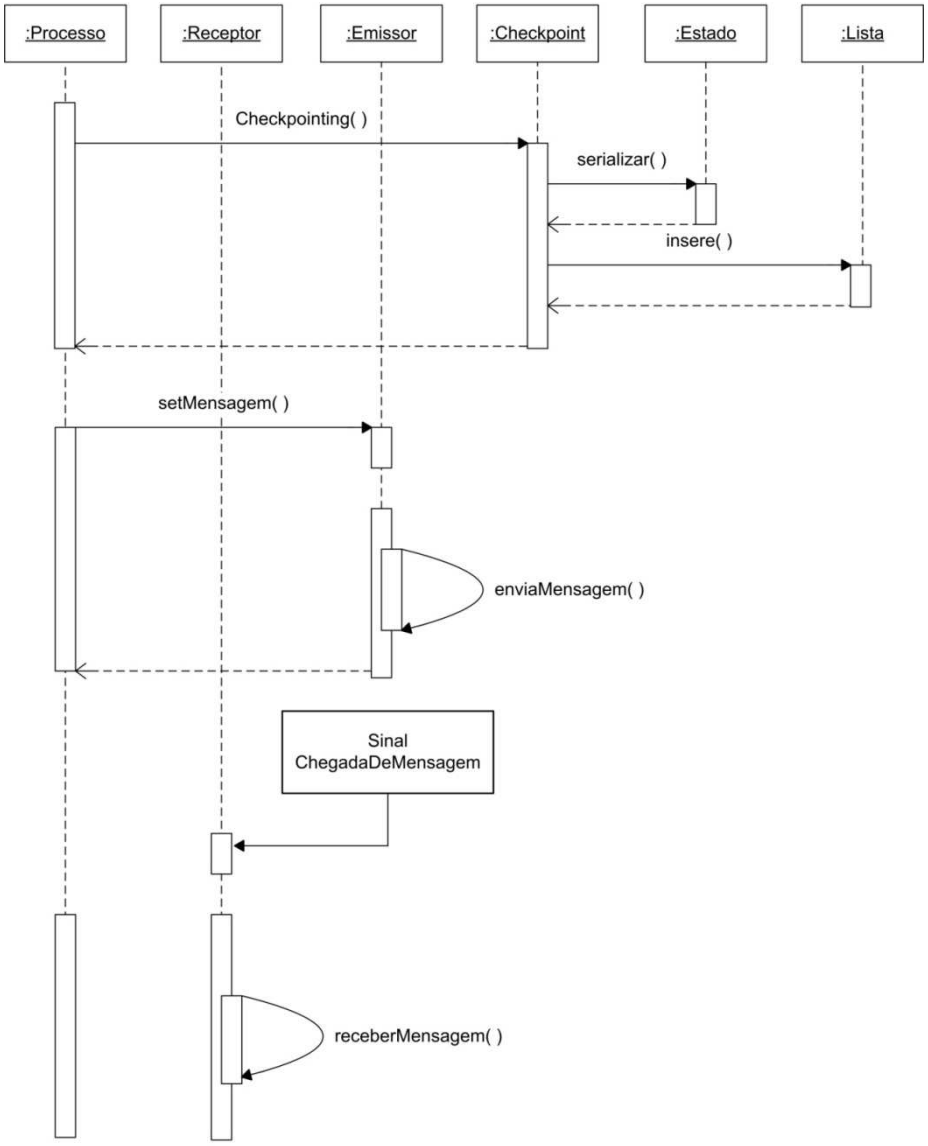

Figura 5.9: Diagrama de seqüência durante um salvamento de estado do protocolo *Time Warp*

Ao ser criado um novo evento, este pode ser interno ou externo. Sendo deste último tipo, o processo deve enviá-lo para ser tratado pelo processo lógico de destino. A mensagem de envio de eventos é construída através do método `setMensagem()` do objeto `Emissor`, que possui a função de transferir o pedido para a rotina responsável na camada de Comunicação. Uma vez construída a mensagem, o objeto `Emissor` chama a rotina de envio de mensagem, ou seja, o método `enviaMensagem()`.

Por fim, o processo lógico deve estar preparado para receber eventos de outros processos através de mensagens. Toda esta rotina esta implementada pelo objeto `Receptor`, que recebe a mensagem através do método `receberMensagem()`.

O objeto `Receptor` também possui a função de identificar se a mensagem recebida é uma mensagem *straggler*. Neste caso, o protocolo deve entrar em procedimento de *rollback* com o propósito de manter a confiabilidade do sistema. A figura 5.10 mostra o comportamento do protocolo quando uma mensagem *straggler* é identificada. Primeiramente, o processo lógico irá identificar o estado salvo com um *LVT* menor ao *timestamp* da mensagem recebida, ação realizada através do método `encontrarEstado()` do objeto `Checkpoint`. Identificado o estado, as informações são obtidas através dos métodos `obterData()` do objeto `Lista` e `desserializar()` do objeto `Estado`. Com as informações necessárias o objeto `Processo` pode proceder à rotina de realização do *rollback*. A última ação a ser realizada pelo o objeto é enviar para os demais processos as anti-mensagens geradas pelo procedimento de *rollback*. Sendo o procedimento de envio e recebimento de mensagens padrão em todos os casos, inclusive no protocolo *Rollback* Solidário, as anti-mensagens serão enviadas da mesma forma que foi apresentado anteriormente.

5.3.2 - Diagramas de Seqüência do protocolo *Rollback* Solidário.

As figuras 5.11 e 5.12 apresentam, respectivamente, o comportamento do protocolo *Rollback* Solidário durante a realização de *checkpoints* básicos ou

forçados e o seu comportamento durante a realização de um procedimento de *rollback*.

Do mesmo modo como ocorre no mecanismo de salvamento de estados *Sparse State Saving*, os *checkpoints* básicos são realizados em intervalos específicos. Além deles, também são realizados os *checkpoints* forçados que geralmente são induzidos pela comunicação.

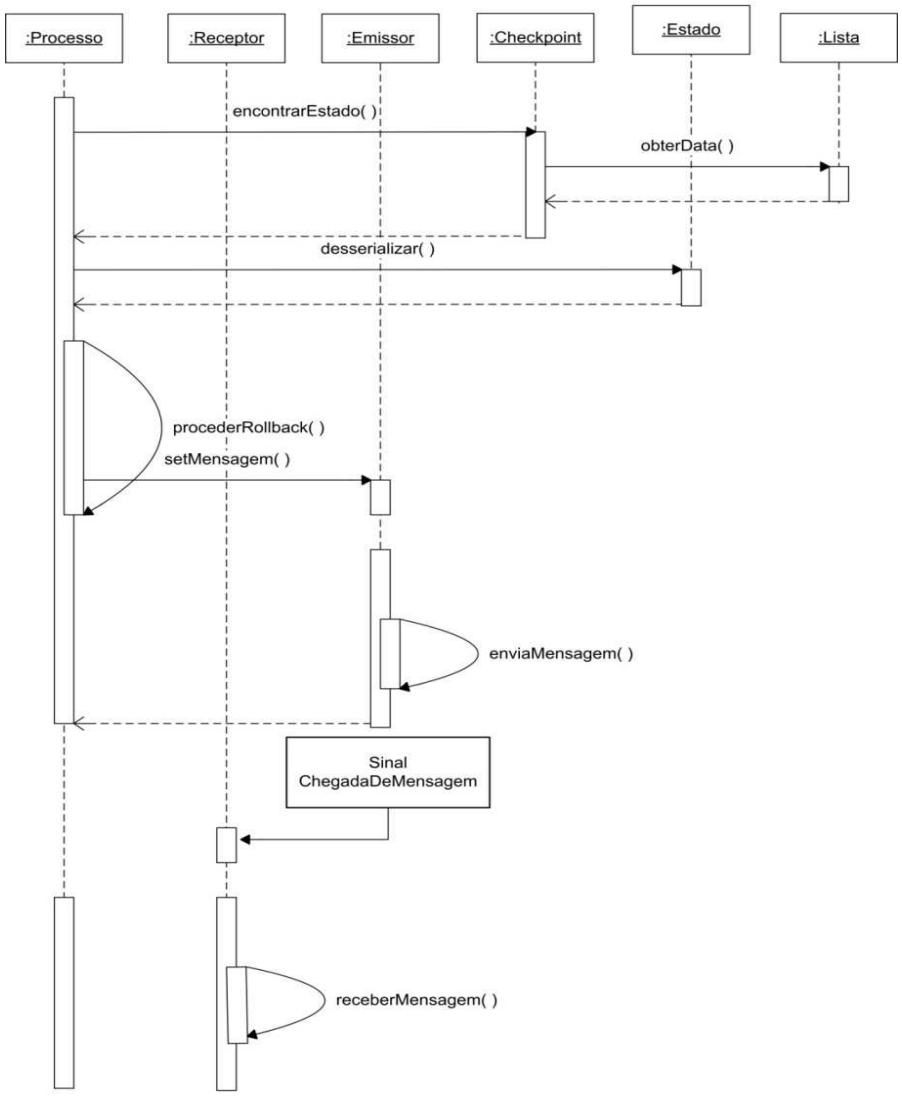

Figura 5.10: Diagrama de seqüência durante um procedimento de *rollback* do protocolo *Time Warp*

O que difere o diagrama da figura 5.11 com o diagrama da figura 5.9 é o objeto da classe `Observador` e a rotina de envio de *checkpoints* para este objeto. Através do método `Checkpointing()` são realizados os *checkpoints*

do objeto da classe `Processo`. A partir deste ponto, todos os procedimentos de armazenamento, de envio e de recebimento de mensagens são iguais aos dois diagramas anteriores.

A ocorrência de um *checkpoint* básico ou forçado deve ser informada ao processo observador. Quando o sinal de uma nova mensagem chega ao objeto Receptor do processo observador, ele imediatamente realiza uma chamada ao método `ExtrairLinhaDeRecuperacao()` do objeto `Observador_SR`, que irá atualizar o atributo `matriz` e verificar se uma nova linha de recuperação pode ser identificada. Neste caso, esta linha de recuperação será inserida na lista `LinhasDeRecuperacao`.

Já na figura 5.12 pode-se observar o comportamento do protocolo *Rollback* Solidário quando o mesmo recebe uma mensagem *straggler*. Neste caso, quando um processo lógico recebe a notificação da chegada de uma mensagem desse tipo, imediatamente após a identificação e restauração de seu estado, ele envia uma mensagem ao processo observador que, por sua vez, irá realizar uma chamada ao seu método `ProcederRollback()`. Este método utiliza-se do método `ObterLinhaDeRecuperacao()` para encontrar o ponto recuperação do sistema e assim realizar os procedimentos de *rollbacks* descritos no capítulo três.

5.4. Mecanismo de Troca Dinâmica de Protocolos

Realizando uma análise comparativa entre os dois protocolos apresentados neste texto, pode-se perceber que a principal diferença está no modo como ambos os protocolos realizam o procedimento de *rollback*. Enquanto o protocolo *Time Warp* utiliza anti-mensagens para propagar a restauração do sistema para todos os processos, o protocolo *Rollback* Solidário utiliza o conceito de *checkpoints* globais consistentes para restaurar todo o sistema de uma única vez.

Observando os diagramas de classes apresentados nas figuras 5.6 e 5.7 verificar-se que além de uma grande semelhança na estrutura dos protocolos, existem elementos comuns entre eles que permitem elaborar um procedimento de troca dinâmica de protocolos durante a execução de uma simulação, bastando para isto, apenas converter os dados da simulação de um protocolo para o outro.

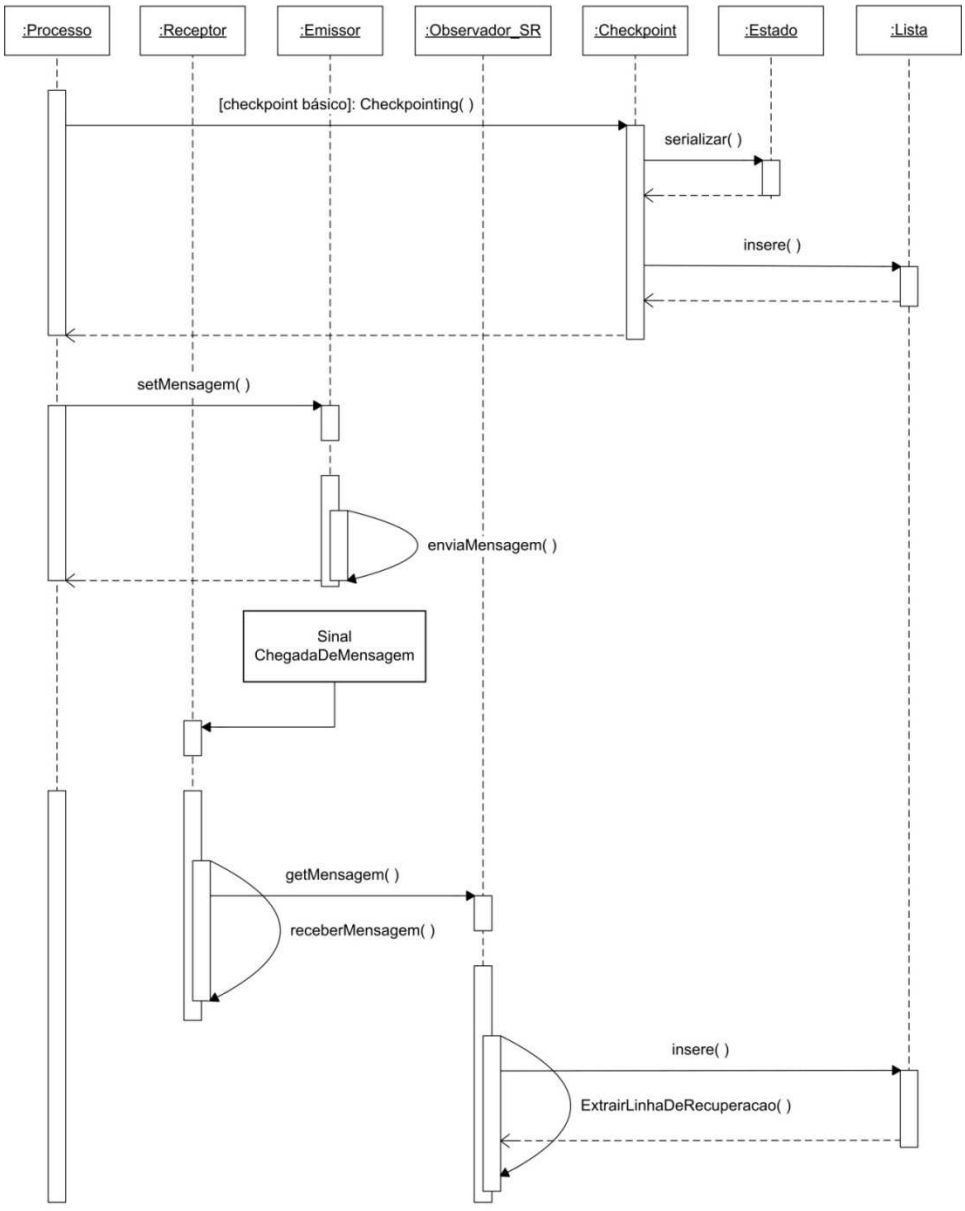

Figura 5.11: Diagrama de seqüência durante um *checkpoint* básico no protocolo *Rollback* Solidário

Para que a troca dinâmica seja realizada com sucesso é necessária a existência de uma classe que decida quando deve ocorrer a mudança dos protocolos. No *framework* apresentado a classe responsável em realizar esta tarefa é a classe `Protocolo`, já discutida neste capítulo. Para também desempenhar esta função, a classe `Protocolo` deve equacionar o problema do desempenho da simulação, por isto, ela deve receber das classes `Observador` de ambos os protocolos as informações referentes à ocorrência de *rollbacks* durante a simulação. Com estas informações, a classe `Protocolo` decide quando deve ser realizada a troca global de protocolos em todos os processos da simulação.

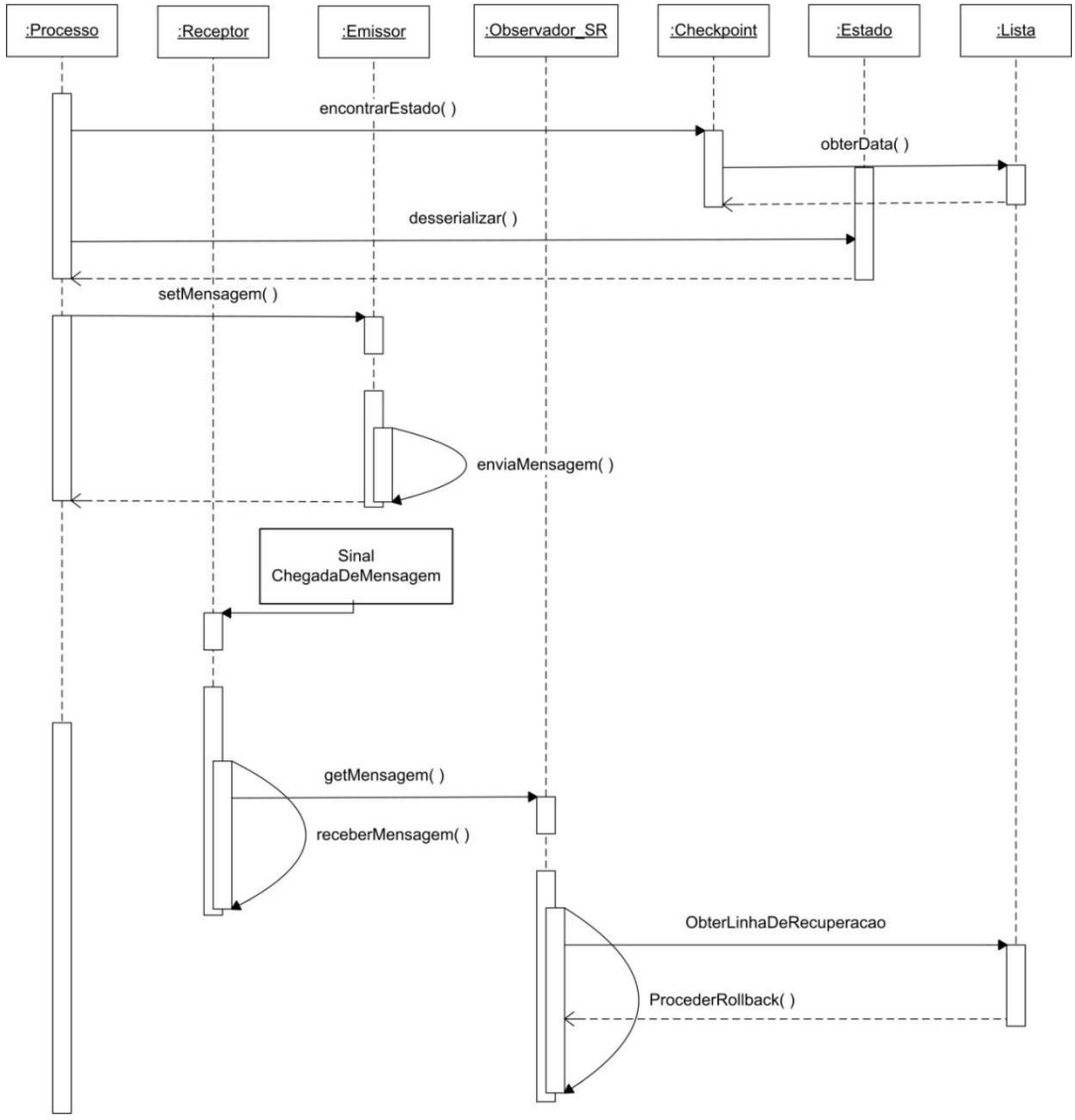

Figura 5.12: Diagrama de seqüência durante um procedimento de *rollback* do protocolo *Rollback* Solidário

A troca deve ser realizada com segurança para garantir que *rollbacks* possam ser realizados corretamente após a mudança de protocolo. Para manter as estruturas de dados válidas para ambos os protocolos uma sugestão seria realizar o armazenamento de anti-mensagens e o procedimento de *checkpointing* continuamente, porém, entende-se que o sistema estaria utilizando os dois protocolos simultaneamente, gerando uma carga excessiva de processamento.

O grande problema para mudar o protocolo é a possibilidade de ocorrência de um *rollback* que force a recuperação do sistema para um ponto da simulação em que o protocolo que tratou tais eventos não seja o protocolo corrente. Como os protocolos em questão tratam erros de causa e efeito de forma diferente entre si, a ocorrência de uma situação assim irá obrigar o protocolo *Time Warp* a realizar um procedimento de *rollback* sem a utilização de anti-mensagens; ou o protocolo *Rollback* Solidário a realizar a mesma ação sem o emprego de linhas de recuperação. Desta forma, para que isto seja possível, será necessária a utilização de um novo mecanismo desenvolvido especificamente para solucionar este tipo de problema.

Para solucionar tal problema a opção encontrada é garantir que todo passado histórico do protocolo antes da troca não será utilizado. E a maneira de atingir parte deste objetivo é utilizando o cálculo do *GVT*, garantindo que nenhum processo da simulação irá requisitar um retorno para algum ponto anterior ao *GVT*. Porém, durante a troca dinâmica de protocolo ainda há o risco dos processos, que possuem os *LVT's* maiores do que o *GVT*, receberem uma mensagem *straggler*. Para esta situação, a solução é permitir que os protocolos coexistam por um determinado período, com o propósito, de fornecer dados suficientes ao novo protocolo responsável pela simulação.

Definir o período de coexistência dos protocolos, *Time Warp* e *Rollback* Solidário, é uma função da classe `Protocolo`. Para isto, deve-se estipular o tempo mínimo necessário para a transição. Assim, quando a classe `Protocolo` decidir trocar os protocolos, ela realiza o cálculo do *GVT* do sistema e obtém o tempo lógico de cada processo. Durante este processo a classe `Protocolo` deve obter o *LVT* máximo do sistema definindo a fronteira

final da simulação híbrida, ou seja, quando o *LVT* máximo for o *GVT* da simulação o novo protocolo poderá assumir a simulação, pois este já possuirá todos os dados necessários para realizar uma recuperação do sistema, se necessário.

A figura 5.13 exemplifica o procedimento de troca de protocolos. A janela de transição representa o período em que os protocolos coexistem. Já a fronteira inicial indica o momento em que a classe `Protocolo` decidiu iniciar a troca do protocolo corrente por um novo protocolo, e por fim, a fronteira final representa o momento em que o *LVT* máximo se tornou o *GVT* da simulação, finalizando o período da simulação híbrida.

Todo gerenciamento da troca dinâmica por parte dos protocolos é realizada pelo processo Observador através da classe `Observador`. Qualquer processo lógico, através da classe `Protocolo`, pode identificar a viabilidade da troca de protocolos. Quando isto ocorre, o processo lógico envia um sinal para o processo Observador solicitando a mudança de protocolo. Neste momento, o processo Observador analisa se o procedimento pode ocorrer ou não. Caso o protocolo corrente seja o *Rollback* Solidário, e o mesmo estiver realizando um procedimento de recuperação do sistema, o processo Observador finalizará o procedimento de *rollback* e realizará a troca dinâmica de protocolos. Caso o protocolo corrente seja o *Time Warp*, o processo Observador verificará se algum processo do sistema está realizando um procedimento de recuperação, ou seja, realizando um *rollback* ou enviando uma anti-mensagem. Neste caso, o processo Observador aguardará o fim do procedimento de recuperação para realizar a troca dinâmica de protocolos. Nos demais casos, quando for solicitada a troca dinâmica de protocolos, o processo Observador irá coordenar a atividade solicitada.

Nas trocas do protocolo *Rollback* Solidário pelo protocolo *Time Warp* o vetor de dependências é descartado, pois não há utilidade para o mesmo no protocolo *Time Warp.* No caso oposto, o protocolo *Rollback* Solidário não receberá um vetor de dependências, contendo dados da simulação, uma vez que o protocolo *Time Warp* não possui esta estrutura de dados. Neste caso, antes de continuar a simulação, o objeto `Observador`, do protocolo *Rollback*

Solidário, irá atualizar os vetores de dependências de todos os processos do sistema com os valores iniciais de uma simulação. Por exemplo: em um sistema que contém três processos, e há a troca do protocolo *Time Warp* pelo *Rollback* Solidário, os vetores de dependência dos processos se iniciarão, respectivamente, por *{1, 0, 0}*, *{0, 1, 0}* e *{0, 0, 1}*.

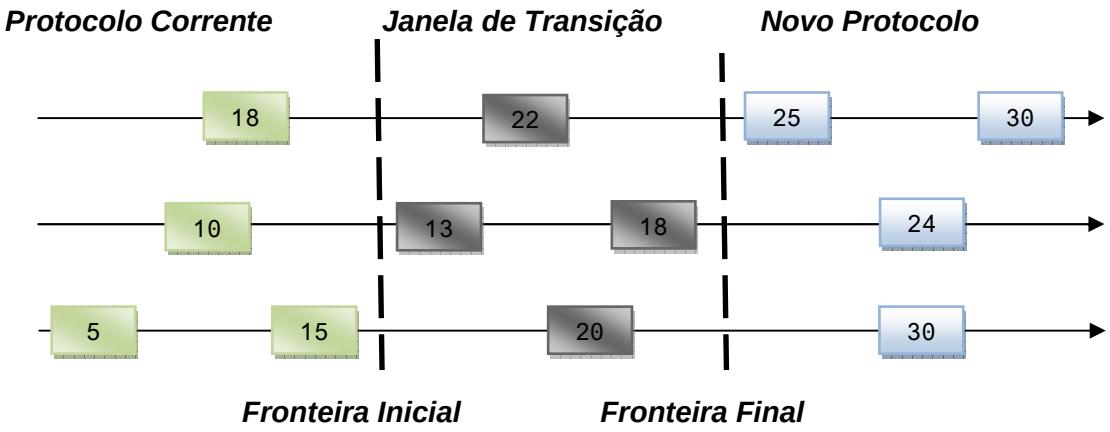

Figura 5.13: Efetivação da troca Dinâmica de Protocolos

Finalmente, é importante considerar que esta abordagem pode ser implementada para quaisquer protocolos, não somente os protocolos *Time Warp* e *Rollback* Solidário.

5.5. Considerações Finais

A implementação do *framework* proposto facilitará de forma considerável a obtenção dos resultados de um modelo através de uma simulação em um ambiente distribuído, pois a sua utilização não exige do usuário um conhecimento profundo de computação distribuída. Para utilizar um ambiente distribuído basta que o usuário tenha um bom conhecimento dos equipamentos que serão utilizados no sistema, do modelo a ser simulado e dos recursos disponíveis pelo *framework*.

As classes que formam o nível Arquitetura, apesar de oferecerem todos os recursos para a implementação da simulação distribuída, apresentam grande funcionalidade e flexibilidade permitindo que alterações e atualizações

possam ser realizadas de acordo com a necessidade do usuário. Todo o sistema pode ser reestruturado para receber novos protocolos de sincronização e bibliotecas de comunicação. Os próprios protocolos *Time Warp* e *Rollback* Solidário podem ser modificados alterando algumas de suas classes. Um exemplo que pode ser citado é a troca do mecanismo de salvamento de estados do protocolo *Time Warp* de *Sparse State Save* para o mecanismo *Copy State Save*, bastando, para isto, criar uma nova classe descendente da classe `Checkpoint`.

As classes abstratas desenvolvidas para o *framework* também oferecem generalidade suficiente para permitir o emprego de classes, como a classe `Comunicacao` e a classe `Mensagem`, em aplicações que necessitem de suporte para troca de mensagens.

Logo, pode-se verificar que o *framework* desenvolvido apresenta alto grau de reutilização e generalização permitindo que usuários com experiência ou não em sistemas distribuídos possam empregar seus recursos em situações diversas. O trabalho apresentado também oferece uma grande quantidade de classes que podem ser empregadas em sistemas que não necessariamente serão utilizados para uma simulação distribuída. Enfim, o *framework* pode auxiliar pesquisadores da área a acelerar a etapa de implementação de seus sistemas, permitindo enfatizar os resultados de suas pesquisas.

Capítulo 6: Discussão sobre a Implementação do *Framework* e Conclusões

Como foi apresentado no capítulo quatro, para se construir um *framework* deve-se, inicialmente, realizar a definição do modelo de objetos do domínio. Durante este projeto, o estudo do protocolo *Rollback* Solidário, foi o ponto de partida para se entender o domínio da aplicação e as necessidades dos usuários para a utilização deste protocolo. As especificações apresentadas em Moreira (2005) foram avaliadas para se definir o que deveria ser fixo e o que deveria ter flexibilidade para que o usuário pudesse realizar modificações de acordo com suas necessidades ou para ampliação dos recursos oferecidos pelo *framework*. Os elementos fixos do *framework* são formados por todas as classes com funções operacionais que possuem a responsabilidade de prover recursos como: interface com o usuário, trocas de mensagens entre os processos e armazenamento de dados. Já os *hot-spots* são formados por componentes que implementam a sincronização dos processos e tratam os erros de causa e efeito. O *framework* possui a flexibilidade para substituir os próprios protocolos de sincronização por outros que possuem a mesma função, ou ser incrementado com outro protocolo. Um exemplo seria acrescentar o protocolo conservativo *CMB* na implementação do *framework*.

Em adição, a comparação entre os protocolos *Rollback* Solidário e *Time Warp* proporcionou identificar os requisitos do *framework*, como: a especificação de quais classes deveriam ser exclusivas ou comuns aos dois protocolos e a implementação das classes `Mensagem` e `Comunicação` com um alto nível de generalização, com o intuito de oferecer condições para transmitir mensagens de tamanhos e estruturas diferentes, de acordo com o protocolo corrente ou com a necessidade do usuário. Iniciou-se, então, a modelagem do sistema, conforme apresentado no capítulo anterior.

6.1. Desenvolvimento dos Modelos do *Framework*

O ponto de partida da fase de modelagem foi a realização de estudos sobre o modelo do protocolo *Rollback* Solidário desenvolvido por Moreira (2005), identificando quais classes poderiam ser utilizadas, modificadas ou retiradas, para atender as especificações do *framework*. A especificação do protocolo *Time Warp* foi desenvolvida empregando algumas classes, que poderiam ser comuns aos dois protocolos, identificadas durante o estudo do protocolo *Rollback* Solidário.

Desenvolvidos os modelos dos protocolos de sincronização, partiu-se para a modelagem da estrutura do *framework*. Nesta etapa, inicialmente foram identificados todos os recursos necessários para que o *framework* pudesse atuar como suporte para ambos os protocolos. Foram modeladas a classe `Protocolo`, para sustentar os protocolos de sincronização; a classe `Mensagem` e `Comunicação`, para dar suporte ao sistema de trocas de mensagens e as ferramentas *PVM* e *MPI*; a classe `Modelo`, empregada para armazenar os dados do modelo a ser simulado; a classe `Arquitetura`, com toda a descrição lógica da arquitetura física do sistema; e, por fim, a classe `Ambiente`, responsável em orquestrar todas as classes apresentadas.

Tendo identificados estes recursos, o desenvolvimento da modelagem do *framework* caminhou para a elaboração das classes que seriam

empregadas como apoio para realizar a interface dos protocolos com o usuário e finalizou com a implementação dos protocolos *Rollback* Solidário e *Time Warp*.

6.2. Implementação do *Framework*

Apesar da especificação desenvolvida, a implementação deste *framework* não é uma tarefa trivial. Isso ocorre devido à união de diversos conceitos como, por exemplo, programação orientada a objetos, sistemas distribuídos, sincronização entre processos distribuídos, linhas de controle *(threads)* e seus respectivos mecanismos de sincronização, bibliotecas de comunicação, além do conhecimento do *hardware* empregado para testes e obtenção de resultados.

O ponto de partida desta fase do trabalho foi a implementação dos protocolos de sincronização: *Time Warp* e *Rollback* Solidário. A biblioteca de comunicação inicialmente adotada para implementar os protocolos foi o *PVM*, utilizando a linguagem de programação C++.

Durante o desenvolvimento dos protocolos alguns problemas foram encontrados e tiveram que ser superados até que os primeiros testes pudessem ser realizados. Inicialmente, todos os computadores da rede do laboratório *LASER* (Laboratório de Segurança e Engenharia de Redes) do Instituto de Engenharia de Sistemas Tecnologias da Informação da Universidade Federal de Itajubá, foram configurados para receber os protocolos. Este laboratório é composto de 10 computadores *Pentium IV* com sistema operacional *Linux*.

Com a rede configurada iniciou-se os testes com o *PVM*, e logo no início da implementação um problema foi identificado: nem todas as mensagens enviadas durante a simulação eram recebidas. Este fato não deveria ocorrer, de acordo com a documentação do *PVM*, uma vez que esta ferramenta garante a ordem de entrega das mensagens. Após vários testes e um estudo mais aprofundado verificou-se que o problema estava na implementação do

programa. Ocorria que o conteúdo do *buffer* de envio da biblioteca era substituído antes do envio da mensagem armazenada. Apesar da sincronização natural existente no uso das primitivas de comunicação *send* e *receive*, foi necessário utilizar semáforos para sincronizar as *threads* dos protocolos, uma vez que as classes `Emissor` e `Receptor` são classes ativas e as respectivas *threads* acessam recursos compartilhados, em especial, a lista de eventos futuros. Assim sendo, além da exclusão mútua, técnicas como às encontradas nos problemas clássicos "Barbeiro Dorminhoco" e "Consumidores e Produtores" (TANENBAUM, 2003) foram utilizadas para o sincronismo das *threads*.

A implementação do *framework* está em fase final de desenvolvimento. O protocolo *Rollback* Solidário está em fase de testes e a implementação do protocolo *Time Warp* ainda necessita de ajustes, uma vez que a primeira versão desenvolvida não possui todos os requisitos da modelagem, tendo sido desenvolvido um protótipo para comparação de desempenho em relação ao protocolo *Rollback* Solidário.

6.3. Principais Contribuições deste Trabalho

O desenvolvimento deste *framework* aumentará o desempenho das aplicações de simulação, pois pesquisadores que não possuem conhecimento na área de sistemas distribuídos poderão realizar simulações distribuídas obtendo resultados com maior eficiência. Além disso, o tempo necessário para a criação deste tipo de aplicação irá diminuir significativamente.

A modelagem desenvolvida neste trabalho oferece vários recursos para implementação de outros *frameworks* para ambientes distribuídos utilizando novos protocolos de sincronização. Em especial, destacam-se a possibilidade de co-existência de estruturas de diversos protocolos, além do suporte as duas bibliotecas de troca de mensagens mais utilizadas: *PVM* e *MPI*.

A preparação do *framework* para a implementação de mecanismos para troca dinâmica entre protocolos e para a migração entre processos são

contribuições importantes, pois estas estruturas permitirão avançar os estudos para a criação de um mecanismo de mapeamento dinâmico dos processos de uma simulação.

6.4. Trabalhos Futuros

Este *framework* é uma ferramenta de apoio aos profissionais que necessitam desenvolver programas eficientes de simulação. Desta forma, a partir dele, vários trabalhos podem ser desenvolvidos, tanto na área de *frameworks* para aplicações distribuídas quanto na área de protocolos de sincronização.

A seguir serão apresentadas propostas de trabalhos futuros que podem ser desenvolvidos a partir dos resultados deste trabalho:

- ✓ Implementação e testes da classe `Comunicação` para utilizar a biblioteca de troca de mensagens *MPI*, uma vez que a implementação atual tem por base a biblioteca de troca de mensagens *PVM*.

- ✓ Considerar a extensão do *framework* para protocolos conservativos. Toda a discussão deste trabalho se concentra nos protocolos otimistas: *Rollback* Solidário e *Time Warp*. Por mais difícil que seja a mudança de um protocolo otimista para outro protocolo otimista, esta tarefa bem mais simples que a mudança entre protocolos de classificação diferente (otimista x conservativos). Entretanto, esta abordagem já foi explorada em outros trabalhos, como por exemplo em Kawabata, 2005, e, desta forma, este *framework* permitirá que mecanismos assim possam ser desenvolvidos de maneira controlada.

- ✓ Estudar e propor padrões de *software* desenvolvidos para aplicações de simulação distribuída. Os padrões de projeto de *software* descrevem soluções para problemas recorrentes no

desenvolvimento de sistemas de *software* orientados a objetos. Os padrões de projeto visam facilitar a reutilização de soluções de projeto, isto é, soluções na fase de projeto do *software*, sem considerar a reutilização de código. Também acarretam um vocabulário comum de projeto facilitando comunicação, documentação e aprendizado dos sistemas de *software*. A principal idéia é que o *framework* desenvolvido possa se tornar um padrão, ou ser feito uma análise comparativa com outros padrões de *software* existentes.

- ✓ Implementação de outros protocolos de simulação ou de melhorias de protocolos existentes e análises comparativas de desempenho entre eles.

- ✓ Realizar um estudo para verificar a generalidade da especificação dos modelos que podem ser descritos no arquivo texto. Principalmente, em relação às distribuições de probabilidade.

- ✓ A troca dinâmica de protocolos pode não proporcionar a eficiência desejada em situações onde houver muitas trocas. Além disso, a mudança pode ser realizada entre protocolos que não aproveitem, da melhor forma, as características do sistema simulado. Desta forma, um estudo mais aprofundado destas características permitirá desenvolver um mecanismo de troca de protocolos eficiente.

6.5. Considerações Finais

Várias pesquisas poderiam ganhar agilidade na obtenção dos respectivos resultados se fosse utilizado processamento paralelo em suas simulações. Porém, a falta de conhecimento nesta área faz com que vários pesquisadores abandonem esta abordagem durante seus trabalhos. Os recursos disponíveis no *framework*, apresentados neste trabalho, auxiliam pesquisadores que necessitam gerar simulações computacionais de grande eficiência, a empregar os conceitos de programação paralela em suas pesquisas.

Diferentemente dos *frameworks* e bibliotecas de classes apresentadas no capítulo 4, o *framework* desenvolvido proporciona soluções encontradas em várias ferramentas distintas. Esta característica facilita sua utilização, pois o usuário pode escolher as ferramentas que possui mais afinidade, tornando a compilação de seu trabalho mais rápida e confiável. Sua estrutura também facilita a adaptação de implementações seqüenciais existentes, em versões paralelas.

Criado um programa, utilizando o *framework*, a sua estrutura pode ser reutilizada apenas adaptando as classes abstratas do *framework* à necessidade da nova aplicação.

O *framework* também oferece a vantagem de ser adaptável a novas ferramentas de comunicação ou até mesmo protocolos de sincronização que venham surgir futuramente. Sua flexibilidade permite atualizações constantes podendo amadurecer de acordo com os avanços nesta área de pesquisa.

Referências Bibliográficas

ALMASI, G. S.; GOTTLIEB A. Highly Parallel Computing. 2nd. ed. Redwood City: The Benjamin Cummings Publishing Company, p. 670, 1994.

ANDERT, G. et al. Object Frameworks in the Taligent OS. In: Compcon Spring '94, Digest of Papers, p. 112-121, Publication Date: 28 Feb-4 Mar 1994.

ASSIS, S. P.; SUZANO, R. Framework: Conceitos e Aplicações. CienteFico, Salvador, v.2, Jun./Dez. 2003

BABAOGLU, O.; MARZULLO, K. Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms. In: MULLENDER, S. (Ed.). Distributed Systems. New York: Addison-Wesley, p. 55–96, 1993.

BADEN, S. et al. Abstract KeLP. In: Proceedings of the 10th SIAM Conference on Parallel Processing for Scientific Computing, 2001.

BAKKEN, D. E. Middleware, Washington State University, 2003.

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

BEGUELIN, A. et al. PVM: Parallel Virtual Machine. A user's Guide and Tutorial for Networked Parallel Computing. The MIT Press, 1994.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. UML - Guia do Usuário. Rio de Janeiro: Editora Campus, 2000.

BROWN, D. L.; HENSHAW, W. D.; QUINLAN, D. J. Overture: An Object-Oriented Framework for Solving Partial Differential Equations on Overlapping Grids. In: Proccedings od SIAM Conference on Object Oriented Methods for Scientific Computing, 1999.

BRYANT, R. E. Simulation of Packet Communications Architecture Computer Systems. Technical Report, MIT-LCS-TR-188, Massachussets Institute of Technology, 1977.

CAI, W.; TURNER, S. J. An Algorithm for Distributed Discrete-Event Simulation – the “Carrier Null Message” pproach. Proceedings of the SCS Multiconference on Distributed Simulation, v. 22, n. 1, p. 3–8, January 1990.

CHAND, K. K. Interoperability and Interchangeability in the Overture Framework. In: Proceedings of the SIAM Conference on Computational Science and Engineering, 2005.

CHANDRA, R.; GUPTA, A.; HENNESSY, J. L. COOL: An Object-Based Language for Parallel Programming. IEEE Computer, v. 27, n. 8, p. 14–26, 1994.

CHANDRA, R. The COOL Parallel Programming Language: Design, Implementation and Performance. Tese (Doutorado) — Computer Science Department, Stanford University, 1995.

CHANDY, K. M.; MISRA, J. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. IEEE Transactions on Software Engineering, SE-5, n. 5, p. 440–452, September 1979.

CHANDY, K. M., AND MISRA, J. Asynchronous Distributed Simulation Via a Sequence of Parallel Computations. Commun. ACM 24,4, p , 198-206, Apr. 1981.

CHWIF, L.; PAUL, R. J.; BARRETTO, M. R. P. Discrete Event Simulation Model Reduction: A Causal Approach. Simulation Modelling Practice and Theory, v. 4, n. 7, p. 930–944, 2006.

DIACONESCU, R.; CONRADI, R. A Data Parallel Programming Model Based on Distributed Objects. In: Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'02), p. 455–460, 2002.

DONGARRA, J. et al. Sourcebook of Parallel Computing. [S.l.]: Morgan Kaufmann, 2003.

ENSLOW, P. H. What is a Distributed Data Processing System. Computer, v. 11, n. 1,p. 13–21, January 1978.

FAYAD, M. E.; SCHMIDT, D. C. Object-Oriented Application Frameworks. Communications of the ACM, V. 40, nº 10, p. 32-38, 1997.

FAYAD, W. E., SCHMIDT, D. C., JOHNSON, R. E. Building Application Framework. ISBN 0471-24875-4, 1999.

FLEISCHMANN, J.; WILSEY, P. A. Comparative analysis of periodic state Saving Techniques in Time Warp Simulators. Proceedings of the 9th Workshop on Parallel and Distributed Simulation, v. 25, n. 1, p. 50–58, July 1995.

FUJIMOTO, R. M. Parallel discrete event simulation. communications of ACM, v. 33, n. 10, p. 31–53, October 1990.

GAFNI, A. Rollback Mechanisms for Optimistic Distributed Simulation Systems. Proceedings of the SCS Multiconference Distributed Simulation, v. 19, n. 3, p. 61–67, 1988.

GERLACH, J.; GOTTSCHLING, P.; DER, U. A Generic C++ Framework for Parallel Mesh Based Scientific Applications. In: Proceedings of the 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'01), p. 45–54, 2001.

ISKRA, K. A.; ALBADA, G. D. V.; SLOOT, P. M. A. Time Warp Cancellation Optimisations on High Latency Networks. Proceedings of the Seventh IEEE International Symposium on Distributed Simulation and Real-Time Applications, p. 128–135, 2003.

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

JOHNSON, R.; FOOTE, B Designing Reusable Classes. Journal of Object-Oriented Programming, New York, v. 1, n. 2, p. 22-35, June/July 1988.

KAWABATA, C. L. O. Gerenciamento do Mecanismo de Troca de Protocolos de Simulação Distribuída em Tempo de Execução. Tese (Doutorado) – USP, São Carlos, 2005.

LAMPORT, L.; SHOSTACK, R.; PAESE, M. The Byzantine Generals Problem. ACM Trans. Programming Languages and Systems, v. 4, n. 3, p. 382–401, 1982.

MARKIEWICZ, M. E.; LUCENA, C. J. P. Object Oriented Framework Development. ACM Crossroads, Volume 7, Issue 4, pp.3-9. , 2001.

MCBRYAN, O. A. An Overview of message passing environments. Parallel Computing, vol. 20, p 417-444, 1994.

MISRA, J. Distributed Discrete-Event Simulation. ACM Computing Surveys, p. 39–65, 1986.

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

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.

NICOL, D. M.; REYNOLDS, P. F. Problem-oriented Protocol Design. Proceedings of the 16th Conference on Winter Simulation, p. 471–474, December 1984.

PERIN FILHO, C. Introdução à Simulação de Sistemas. Editora da Unicamp, 1995.

PREE, W. Framework Patterns, New York: SIGS Books, 1996.

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.

QUAGLIA, F.; CORTELLESSA, V. Rollback-based Parallel Discrete Event Simulation by Using Hybrid State Saving. Proceedings of the 9th European Simulation Symposium, p. 275–279, October 1997.

RAJAEI, H. Local Time Warp: An Implementation and Performance Analysis. 21st International Workshop o Principles of Advanced and Distributed Simulation (PADS'07), 2007 IEEE.

REIHER, P. L. et al. Cancellation Strategies in Optimistic Execution Systems. Proceedings of the 1990 Distributed Simulation Conference, v. 22, p. 112–121, January 1990.

REYNOLDS, P. F. A Shared Resource Algorithm for Distributed Simulation. Proceedings of the 9th Annual Symposium on Computer Architecture, p. 259–266, April 1982.

RUSSEL, K.; MADINA, D. ClassdescMP: Easy MPI programming in C++. In: AL, S. et (Ed.). Computacional Science. [S.l.]: Springer-Verlag, (LNCS 2660) 2003.

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

SCHROEDER, M. D. A State-of-the-art Distributed System: Computing with Bob. S.J. MULLENDER (Ed.). In Distributed Systems. 2nd ed. ACM Press, 1993.

SIMONE, M. D. Active Expressions: A Language-Based Model for Expressing Concurrent Patterns. Dissertação (Mestrado) — University of Waterloo, 1997.

SONODA, E.; TRAVIESO, G. The OOPS Framework: High Level Abstractions for the Development of Parallel Scientific Applications. In: OOPSLA'06: Companion to the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications. New York, NY, USA: ACM Press, p. 659–660, 2006.

SQUYRES, J. M.; MCCANDLESS, B. C.; LUMSDAINE, A. Object Oriented MPI: A Class Library for the Message Passing Interface. In: Proceedings of the Parallel Object-Oriented Methods and Applications (POOMA'96), 1996.

STEINMAN, J. S. Incremental State Saving in Speedes Using C++. Proceedings of the 1993 Winter Simulation Conference, p. 687–696, December 1993B.

SUNKPHOT, J. A Framework for Developing Field Inspection Support Systems. Tese (Ph. D.) – B. Eng, Chulalongkorn University, Thailand e M.S.C.E., Georgia Institute of Technology, 2001.

TANENBAUM, A. S. Distributed Operating Systems. Upper Saddle River: Prentice Hall, p. 648, 1995.

TANENBAUM, A. S.; Sistemas Operacionais Modernos. 2ª Edição. São Paulo Pearson – Prentice Hall, 2003.

TEO, Y. M.; NG, Y. K. Spades/java: Object-Oriented Parallel Discrete-Event Simulation. Proceedings of the 35th Annual Simulation Symposium, p. 245–252, 2002.

WALKER, D. W. An MPI Version of the BLACS. In Proceedings of the 1994 Scalable Parallel Libraries Conference, Oct, 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.

WANG, J; TROPPER, C. Optimizing Time Warp Simulation with Reinforcement Learning Techniques. Proceedings of the 2007 Winter Simulation Conference p. 577-584, 2007.

WISSINK, A. M. et al. Large Scale Parallel Structured AMR Calculations Using the SAMRAI Framework. In: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing SC'01. 2001.

WISSINK, A.; HYSOM, D.; HORNUNG, R. D. Enhancing Scalability of Parallel Structured AMR Calculations. In: Proceedings of the 17th ACM International Conference on Supercomputing (ICS03). p. 336–347, 2003.

ZENG, Y.; Cai, W. e Turner S. J. Batch Based Cancellation: A Rollback Optimal Cancellation Scheme in Time Warp Simulation. 18th Workshop on Parallel and Distributed Simulation (PADS'04), IEEE, 2004,

ZIMMERMAN, A.; KNOKE, M.; HOMMEL G. Complete Event Ordering for Time Warp Simulation of Stochastic Discrete Event Systems. 4th Symposium on Design, Analysis, and Simulation of Distributed Systems (DASD 2006).