UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Proposta de um suporte para simulações orientadas a objetos distribuídas em uma plataforma aberta

Dissertação submetida à Universidade Federal de Santa Catarina
como requisito parcial à obtenção do grau de

Mestre em Engenharia Elétrica

por

Richard Duarte Ribeiro

Florianópolis, 09 de junho de 1997

Proposta de um suporte para simulações orientadas a objetos distribuídas em uma plataforma aberta

Richard Duarte Ribeiro

Esta dissertação foi julgada para a obtenção do título de **Mestre em Engenharia**, especialidade **Engenharia Elétrica**, área de concentração **Sistemas de Controle, Automação e Informática Industrial**, e aprovada em sua forma final pelo curso de Pós-Graduação da Universidade Federal de Santa Catarina.

Prof. Carlos Alberto Maziero, Dr.
Orientador

Prof. Adroaldo Raizer, Dr.
Coordenador do Curso de Pós-Graduação

BANCA EXAMINADORA

Prof. Carlos Alberto Maziero, Dr.

Prof. Jean-Marie Farines, Dr.

Prof. Joni da Silva Fraga, Dr.

Prof. Luiz Nacamura Jr., Dr.

À Rosamelia, minha companheira nessa batalha e em muitas outras ...

Agradecimentos

Ao CEFET-PR e à CAPES-PICDT, pelo suporte financeiro.

Ao LCMI na UFSC pela oportunidade concedida.

Aos professores do DAINF no CEFET-PR pelo apoio e compreensão.

Ao professor Carlos Alberto Maziero, pela orientação deste trabalho, pela amizade e pela confiança depositada.

Aos membros da banca examinadora pela participação, críticas e comentários.

A Lau Cheuk Lung, Damián Rodrígues Sánchez, Nelkis de la Orden Medina pela ajuda, orientação e colaboração.

A José Aguiomar Foggiatto, pelas orientações antes, durante e depois do trabalho.

A todos que direta ou indiretamente ajudaram na elaboração deste trabalho.

A Deus.

Resumo

A simulação é uma ferramenta de grande utilidade no estudo de sistemas complexos. As técnicas de simulação têm evoluído no sentido de aproveitar os avanços constatados em outras áreas da informática, particularmente no campo das linguagens de programação, para facilitar a construção dos modelos, e do processamento paralelo, para agilizar a execução de simulações de grandes dimensões.

Este trabalho visa definir a estrutura de um suporte de execução para simulações a eventos discretos distribuídas. Para simplificar a construção dos modelos de simulação, optou-se pelo uso de linguagens com características de orientação a objetos, tornando possível a construção hierárquica dos modelos e a reutilização de código. Ao invés de criar uma linguagem de descrição de modelos própria para o suporte proposto, optou-se por empregar bibliotecas conhecidas de simulação orientadas a objetos, obtendo assim um suporte mais genérico e portável.

Os problemas provenientes do uso da programação orientada a objetos sobre um suporte computacional distribuído e possivelmente heterogêneo são abordados através do uso de uma plataforma aberta para a comunicação entre objetos, seguindo o padrão CORBA.

Abstract

Simulation is an useful tool in the study of complex systems. Traditional simulation techniques have been improved by using developments achieved in other computer science domains, like programming languages and parallel processing. Construction of simulation models is going easier, by means of new programming concepts, and huge simulations can run faster, due to parallel processing techniques.

This work aims to define an open platform structure for distributed discrete-event simulations. To make the construction of simulation models easier, we adopted an object-oriented modelling approach, allowing hierarchical models and code reuse. In place of defining our own object-oriented simulation language, we choosed to use known object-oriented simulation libraries, to obtain a more generic and portable simulation environment.

The problems arising from the use of an object-oriented language over a distributed and maybe heterogeneous execution environment have been adressed by using an open platform for objects interactions, matching the CORBA standards.

Índice

Lista de Figuras

Capítulo 1

Introdução

O estudo de grandes sistemas pode ser efetuado através de um enfoque analítico ou via simulação por computador. Esta última forma é particularmente interessante no caso de sistemas complexos ou com muitas entidades, dificultando uma abordagem analítica [CM81, RW89, Maz94].

Todavia a simulação de sistemas complexos pode ser uma tarefa pesada, em termos de esforço computacional. Por exemplo, a simulação do funcionamento de um comutador telefônico de grande porte durante alguns minutos pode consumir semanas de processamento [Mis86].

Nestes casos a paralelização da simulação pode ser de grande utilidade, haja visto a significativa quantidade de paralelismo potencial existente nesse tipo de sistema. Entretanto, paralelizar de modo eficiente uma simulação sem pôr em risco sua propriedade fundamental, o respeito à causalidade no sistema modelado, não é uma tarefa trivial, e tem sido alvo de pesquisa nos últimos anos [Mis86, RW89, Fuj90, Maz94].

A abordagem de programação por objetos pode ser extremamente útil na construção de modelos de simulação complexos, por permitir uma tradução mais intuitiva das entidades do sistema e suas interações para um modelo computacional. O próprio paradigma de programação orientada a objetos tem suas origens na simulação, através da linguagem Simula 67, que introduziu os conceitos de objetos, atributos e métodos [BDMN73].

Este trabalho tem como objetivo propor uma estrutura para suportar a execução distribuída de simulações orientadas a objetos sobre plataformas computacionais possivelmente heterogêneas. Para permitir a interação entre os diversos objetos que compõem o simulador distribuído e o modelo a simular, que podem estar em máquinas e sistemas operacionais distintos, utilizamos um suporte CORBA [Obj95, Siq95].

Esta dissertação está dividida em seis capítulos, descritos a seguir:

Capítulo 1 Contém uma breve introdução à dissertação, situando o contexto em que a

mesma se encontra. São apresentados os objetivos que se prentende alcançar com este trabalho. Além disso, descreve-se de forma sucinta o conteúdo deste trabalho, procurando ressaltar seus pontos mais importantes.

Capítulo 2 Apresenta os principais conceitos e técnicas empregadas em simulação seqüencial. Apresenta também a problemática ligada à paralelização de uma simulação, discutindo seus principais problemas e as técnicas usadas para solucioná-los.

Capítulo 3 Introduz o uso da orientação a objetos na simulação. Aqui são explicadas resumidamente quais ferramentas são usadas neste tipo de simulação (linguagens específicas e bibliotecas especializadas). A seguir é explicada a biblioteca especializada utilizada por nós neste trabalho, a biblioteca C++SIM.

Capítulo 4 Neste capítulo é apresentada a arquitetura CORBA. É explicado como esta funciona e qual a sua estrutura. A seguir é apresentado o núcleo CORBA utilizado neste trabalho, o CHORUS/COOL.

Capítulo 5 Neste capítulo apresentamos a estrutura proposta para a simulação distribuída de modelos construídos segundo o paradigma de orientação a objetos. São abordadas a estrutura local a cada máquina e as interações entre as máquinas que compõem o suporte de execução.

Capítulo 6 Este finaliza o trabalho, mostrando os resultados obtidos com este trabalho e as conclusões obtidas com o mesmo. São propostas também algumas idéias para trabalhos futuros que poderão complementar e engrandecer o presente trabalho.

Capítulo 2

Simulação a Eventos Discretos

2.1 Introdução

Nos dias de hoje a existência de grandes sistemas complexos já se tornou um fato comum. Esse tipo de sistema encontrado em diversas áreas do conhecimento humano (economia, telefonia, manufatura, entre outras) requer um cuidadoso estudo para o controle e acompanhamento de seus comportamentos, bem como para medidas de desempenho e verificação de possíveis modificações. Isso tudo pode se mostrar inviável se pensarmos nas limitações encontradas pelos métodos matemáticos convencionais. Uma solução possível para conseguir as referidas medições e estudos é a simulação.

A metodologia do estudo de um sistema através da simulação é simples: com base em um modelo simplificado do sistema real, pode-se fazer a sua observação no tempo e realizar os estudos que se fizerem necessários. Tal modelo deve reproduzir o comportamento do sistema real com a exatidão requerida para os estudos.

Para se conseguir êxito na simulação deve-se obter, a partir do sistema ou subsistema em estudo, um modelo que o represente segundo os aspectos mais relevantes do comportamento em estudo. Com base na complexidade imposta pelo modelo escolhe-se qual técnica de simulação será utilizada para analisá-lo. No fim obteremos um modelo de simulação, ou seja, um programa de computador que contém a lógica operacional do sistema ou subsistema em estudo [PHB93].

Em uma simulação o tempo pode ser contado de uma forma independente do tempo real. Este tempo está vinculado a simulação e é chamado de "tempo simulado" ou "tempo lógico", ou ainda "tempo virtual". Isto permite que a simulação seja executada seguindo uma velocidade controlada, possibilitando uma evolução do sistema compatível com a observação e estudo do mesmo.

2.2 Princípios dos Sistemas Físicos

Todo sistema físico obedece a dois príncipios básicos, cujo respeito deve ser garantido durante a simulação de um modelo do mesmo:

- *Causalidade.* Em um dado instante t o sistema possui um estado que é dependente do seu estado inicial, do próprio tempo t, e dos estados anteriores até o momento t (incluindo este) [Mis86, Maz94]. Isso significa que o futuro não pode influenciar o passado. Este princípio deve ser garantido pelo simulador.

- *Determinismo.* A todo instante t existe um valor positivo ϵ, tal que o comportamento futuro do sistema pode ser completamente determinado até $t + \epsilon$ [Mis86]. Tal propriedade deve ser corretamente mapeada no modelo construído. Este princípio é de responsabilidade da modelagem do sistema.

2.3 Modelos de Simulação

Um sistema pode ser visto como uma coleção de entidades que interagem entre si. Tais entidades representam as características de funcionamento do sistema. Cada entidade pode ser descrita com o auxílio de variáveis de estado e de funções que cuidam da evolução dessas variáveis. Tais variáveis e funções se apresentam sob formas diferentes, dependendo do tipo de sistema a modelar. Existem duas abordagens que podem ser usadas para a modelagem de sistemas:

- *Sistemas contínuos*: neste caso a evolução do sistema é melhor representada de forma contínua. Para descrever esse tipo de comportamento a modelagem do sistema é feita através de um conjunto de equações diferenciais, montado sobre as variáveis de estado, que assim evoluem de forma contínua.

- *Sistemas discretos*: a evolução do sistema é melhor modelada através da ocorrência de eventos em instantes discretos. Esse comportamento pode ser representado através de transições de estado (eventos) que possuem pré e pós condições aplicadas sobre as variáveis de estado.

Por questão de simplicidade, neste trabalho consideraremos apenas os sistemas discretos.

2.4 Simulação a Eventos Discretos

2.4.1 Evolução do Tempo Simulado

Na execução de uma simulação deve-se escolher como o tempo simulado evoluirá. Levando-se em conta que sempre deverão ser respeitados os dois princípios básicos dos sistemas físicos (causalidade e determinismo) existem duas formas básicas de fazer evoluir o tempo [RW89]:

- *Simulação coordenada pelo tempo.* Neste tipo o tempo evolui em passos fixos (*ticks*), de duração δ. Depois de cada passo, o simulador verifica a existência de eventos que possuam data de ocorrência igual à data atual no tempo simulado e os executa. Após isso, o simulador avança o tempo simulado em mais um passo. O grande problema desse tipo de simulação reside na escolha que deve ser feita do valor de δ. Valores grandes levarão à perda de eventos, tornando a simulação errônea. Valores pequenos levarão à perda de eficiência, pois resultarão em vários passos sem a ocorrência de algum evento. Essa forma de simulação se torna interessante quando a densidade e espaçamento dos eventos no tempo é regular [Maz94].

- *Simulação coordenada pelos eventos.* Nesta técnica o tempo simulado é avançado do tempo de ocorrência de um evento para o tempo de ocorrência do próximo evento. Com isso não se permite a ocorrência de etapas inúteis, já que após cada evolução no tempo simulado existirá ao menos um evento a ser tratado.

No presente trabalho optamos pela simulação coordenada pelos eventos, pois esta é geralmente mais eficiente e permite aproveitar melhor o paralelismo potencial em modelos de grandes dimensões, devido ao seu maior assincronismo.

2.4.2 Estrutura dos Modelos

Para permitir a execução de uma simulação em um computador é necessário traduzir os aspectos estruturais e dinâmicos do sistema em estudo para um modelo de simulação.

Existem várias abordagens utilizadas para realizar essa tradução. Tais abordagens procuram descrever as mudanças de estado e as suas ações resultantes. As principais são [Maz94]:

- *Orientadas a eventos.* Neste caso cada evento no sistema é representado por uma condição para sua ocorrência e pelas ações que serão executadas quando dessa ocorrência. Tais ações são consideradas de duração nula no tempo simulado.

- *Orientadas a processos.* Aqui cada entidade do modelo tem sua representação através de um processo, que executa ações representando as mudanças de estado, o avanço do tempo simulado e as relações com as outras entidades do sistema. Alguma dessas ações são de duração não nula no tempo simulado. As interações entre processos podem ser feitas por recursos (objetos comuns acessados por vários processos), por ativação e desativação direta de processos entre si, ou por troca de mensagens.

Neste trabalho optamos por utilizar o modelo discreto de simulação usando uma abordagem de modelagem por processos e mensagens. Fizemos estas escolhas por achar que este modelo é o que mais se adapta ao tipo de estudo realizado nesta dissertação, além de ser o mais adequado à execução em uma plataforma distribuída.

2.4.3 Mecanismo Básico de Simulação

Na simulação a eventos discretos devemos controlar a ordem dos eventos a serem tratados. Isto deve ser feito para se garantir o respeito à causalidade (garantir que nenhum evento será executado antes de outro com data de execução anterior). Para isso utiliza-se o "escalonador", que pode ser implementado através de uma fila, na qual os eventos a processar são ordenados de forma crescente segundo a data de ocorrência de cada um deles.

A figura 2.1 mostra o esquema básico de um escalonador. O primeiro evento da fila será o próximo evento a ser tratado. A data de ocorrência deste evento corresponde ao instante presente no tempo simulado. O tratamento deste evento pode gerar novos eventos, que por sua vez são colocados no escalonador (mantendo ainda a ordenação crescente). Assim sendo, os outros eventos do escalonador são considerados potenciais, pois podem ser modificados ou cancelados no tratamento dos eventos anteriores.

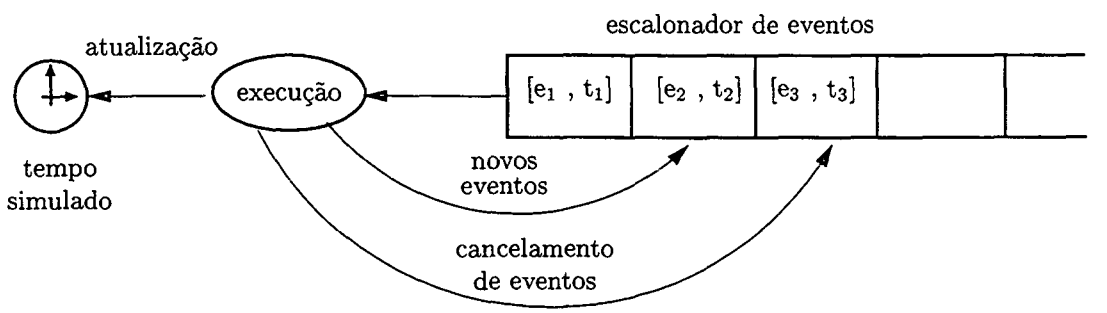

Figura 2.1: Um escalonador.

O esquema de execução definido pelo escalonador garante que todo evento será tratado depois dos eventos dos quais ele pode depender causalmente, pois estes têm uma data de ocorrência anterior à do evento em questão.

Esse mecanismo tem utilidade tanto na simulação orientada a eventos quanto na orientada a processos. No primeiro caso, todos os eventos do sistema são gerenciados por um escalonador único. No segundo caso, cada processo pode manter um escalonador interno para gerenciar a execução de seus eventos locais.

2.5 Simulação Distribuída

A simulação seqüencial de modelos de grandes dimensões pode exigir um esforço computacional considerável, que pode facilmente significar dias ou mesmo semanas de processamento. Entretanto, nesses sistemas existe normalmente uma quantidade de paralelismo potencial significativo, gerado por atividades com baixo grau de dependência, em setores distintos do sistema.

A distribuição da simulação sobre um conjunto de processadores pode desta forma aumentar em muito a velocidade de execução do simulador, mas os mecanismos de simulação e de sincronização entre processos devem ser capazes de explorar esse paralelismo potencial de forma eficiente.

O problema central na execução distribuída de uma simulação é o compromisso permanente entre o aproveitamento máximo do paralelismo potencial do modelo e o respeito ao princípio de causalidade, essencial a qualquer simulação. Diversos trabalhos foram efetuados na busca de soluções a esse problema [CM79, CM81, Mis86, RW89, Fuj90, Maz94]. As principais soluções serão abordadas na seqüência deste capítulo.

2.5.1 Estrutura dos Modelos

Para melhor apresentar os mecanismos de sincronização empregados em simulações distribuídas, vamos primeiramente definir uma estrutura básica para os modelos de simulação considerados. Usando o paradigma processo-mensagem, podemos definir um modelo como sendo constituído por um conjunto de processos que comunicam entre si por mensagens trocadas através de uma rede estática de canais FIFO (figura 2.2):

- *Rede estática de processos*: cada processo modela uma entidade do sistema. Para simplificar o estudo dos algoritmos de sincronização envolvidos, a topologia do modelo é estática (o número de processos é constante e os canais de comunicação entre processos são fixos).

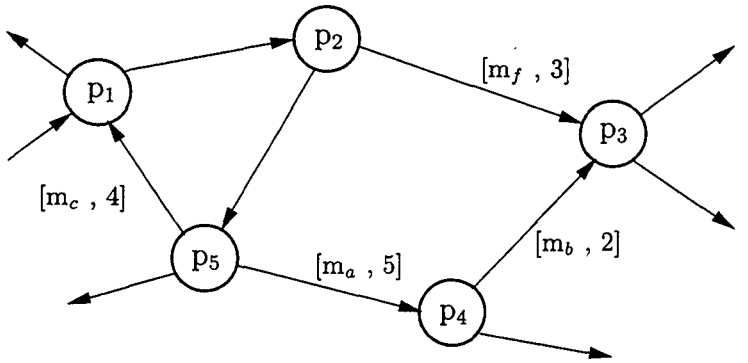

Figura 2.2: Modelo de simulação.

- *Comunicação por troca de mensagens*: iremos modelar as interações entre entidades no sistema real através de trocas de mensagens entre os respectivos processos no modelo. Como no sistema real as interações entre as entidades ocorrem de forma instantânea, consideramos que o tempo de transferência das mensagens entre processos é nulo no tempo simulado. Isso significa que uma mensagem que sai de um processo no instante t chega obrigatoriamente no mesmo instante t ao processo receptor.

- *Canal FIFO ilimitado entre processos*: para as trocas de mensagens os processos usam canais de comunicação unidirecionais, FIFO (*First In First Out*) e ilimitados. Essas características asseguram que as mensagens entre dois processos quaisquer circularão dentro de uma ordem correta. Os canais são não-limitados para evitar que bloqueios ocorram na transmissão de mensagens.

- *Referência temporal única*: no sistema real todas as entidades possuem uma referência temporal única, logo, esta característica deve existir na simulação também. Na simulação seqüencial clássica isso é mantido de uma forma simples[1], mas no caso da simulação distribuída o controle pode ser mais complexo, como veremos a seguir.

2.5.2 Gestão do Tempo Simulado

A questão mais importante da simulação a eventos discretos distribuída é a que se refere ao problema do respeito à causalidade. Na simulação seqüencial, um relógio único é usado para coordenar a seqüência em que serão tratados os eventos do sistema, garantindo

[1]Em um simulador seqüencial o andamento do tempo simulado é representado por um relógio atualizado antes de cada tratamento de evento. Este relógio serve de referência temporal única a todos os processos da simulação [Maz94].

desta forma a causalidade. Na simulação distribuída a evolução do tempo simulado deve também seguir uma referência única, mas o contexto paralelo permite que cada processo se comporte como um simulador seqüencial quase independente. Desta forma podemos vislumbrar duas estratégias distintas para a evolução do tempo simulado em uma simulação distribuída:

- *Abordagem síncrona.* Todos os processos do modelo avançam de maneira síncrona no tempo simulado. Para isso, cada processo determina a data de seu próximo evento a tratar e a comunica a um processo controlador, que calcula o mínimo entre todas as datas recebidas e envia esse dado de volta para os processos. Com esse valor os processos podem avançar no tempo simulado até um tempo mínimo seguro [Maz94]. A implementação do relógio global pode ser centralizada (um processo dedicado atuando como sincronizador), ou distribuída (algumas propostas podem ser encontrada em [RW89]).

- *Abordagem assíncrona.* Cada processo gerencia uma cópia local do relógio de simulação, chamada *relógio local*, que evolui em função do estado local do processo [RW89]. Admite-se assim uma evolução assíncrona dos relógios locais dos processos, e portanto podem ocorrer defasagens entre eles, no tempo simulado. Nesse contexto assíncrono, mecanismos especiais devem ser empregados para gerenciar o tempo simulado e garantir o princípio de causalidade. Além disso, toda mensagem enviada por um processo deve carregar sua data de envio (estampilha).

No nosso trabalho optamos pela abordagem assíncrona, pois esta permite soluções totalmente distribuídas (sem um controle centralizado). Além disso, as soluçõs assíncronas possuem normalmente maior desempenho permitindo explorar melhor o paralelismo potencial do modelo [RW89].

2.6 Garantir a Causalidade

Dentro da simulação assíncrona deve-se garantir uma evolução correta do tempo simulado, sem que ocorra alguma violação à lei da causalidade. Garantir uma evolução correta do tempo simulado e o respeito ao princípio de causalidade em um modelo distribuído no qual os relógios locais dos processos podem evoluir de maneira assíncrona é, à primeira vista, uma tarefa complicada. No entanto, no artigo [CM81] Chandy e Misra demonstram que, se cada processo preservar localmente o princípio de causalidade e os canais de comunicação forem FIFO, então o princípio de causalidade será respeitado pela simulação como um todo.

Para preservar a causalidade localmente, cada processo deve tratar todos os seus eventos locais (eventos internos ou mensagens recebidas de outros processos) na ordem estrita de suas datas de ocorrência no tempo simulado. Por exemplo, na figura 2.3, o processo p_i deve tratar as mensagens recebidas em seus canais de entrada na ordem [m_4 , 3], [m_1 , 4], [m_2 , 7] e [m_3 , 9]:

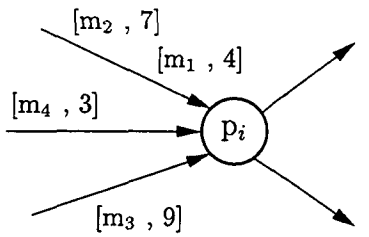

Figura 2.3: Respeito local à causalidade.

Entretanto, quando houverem canais vazios (sem mensagem recebida) na entrada de um processo, este pode encontrar dificuldade para estabelecer a ordem correta das mensagens a consumir, pois novas mensagens podem chegar nos canais vazios. No exemplo da figura 2.4, o processo p_i não pode decidir sobre a ordem de consumo das mensagens presentes em suas entradas, pois uma nova mensagem com estampilha qualquer pode chegar no canal vazio.

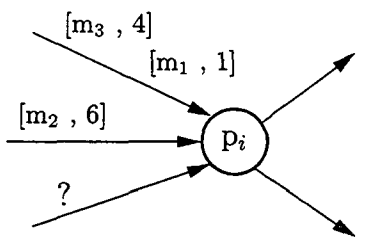

Figura 2.4: Processo em espera de mensagem.

Para resolver esse problema duas classes de estratégia foram propostas [Fuj90]:

- *Estratégia pessimista*: busca-se garantir, *a priori*, que as mensagens serão sempre consumidas na ordem de suas estampilhas, ou seja, o princípio de causalidade nunca é violado. Assim, a execução do processo pode ser suspensa até que a definição inequívoca da próxima mensagem a ser consumida possa ser feita. Mecanismos adicionais podem ser necessários para evitar a ocorrência de bloqueios, como veremos na seção 2.6.1.

- *Estratégia otimista*: aqui as mensagens já disponíveis nos canais de entrada de um processo são consumidas na ordem de suas estampilhas, mesmo se esta ordem não for definitiva. Caso mais tarde cheguem mensagens com estampilhas menores que aquelas já tratadas (violação local do princípio da causalidade), a simulação do processo deve ser refeita a partir daquele ponto, de forma a considerar as mensagens que chegaram atrasadas.

2.6.1 Estratégias Pessimistas

Primeiramente propostas em [CM79], essas soluções são também conhecidas como *mecanismos conservativos* ou técnicas de *correção a priori*. Elas tentam assegurar quando o processamento de um evento será seguro, isto é, caso um processo possua um evento não processado em algum dos seus canais de entrada, ele possa chegar a conclusão em um tempo finito de que pode tratar esse evento sem risco de violar o princípio de causalidade, pois estará seguro de que não chegará a ele outro evento com estampilha menor [Fuj90].

Relógio do Canal e Relógio de Entrada

Conforme visto, a transferência de mensagens através de um canal obedece uma ordem FIFO, que permite estabelecer uma ordem crescente de estampilhas neste. Assim, a estampilha da última mensagem recebida através de um canal estabelece uma data mínima para a próxima mensagem que atravessar esse canal. Esta data mínima é chamada de "relógio do canal" (*channel time*).

Um processo pode determinar a estampilha mínima da próxima mensagem que receberá com base nos relógios de seus canais de entrada. O mínimo entre todos os relógios dos canais de entrada define uma grandeza que chamaremos de "relógio de entrada" (*input clock*) do processo.

Todas as mensagens presentes na entrada de um processo com uma estampilha igual ou inferior ao seu relógio de entrada podem ser consumidas pelo processo sem risco de violação do princípio de causalidade. Caso nenhuma mensagem da entrada possa ser consumida, o processo deve aguardar a evolução de seu relógio de entrada, que depende da evolução dos relógios dos canais de entrada, e portanto da chegada de novas mensagens.

O Risco de Bloqueios

O controle do consumo de mensagens através dos relógios dos canais de entrada permite garantir o respeito ao princípio de causalidade, mas ele gera outro problema: a possibilidade de bloqueio entre processos.

Tal situação ocorre quando se forma um ciclo de processos em espera, cada um aguardando uma mensagem de seu respectivo canal de entrada que possui o menor relógio de canal. A figura 2.5 ilustra uma situação de bloqueio típica (onde rl_i representa o relógio local do processo i, [m_i , t] representa uma mensagem i com estampilha t, e rc_i representa o relógio do canal i):

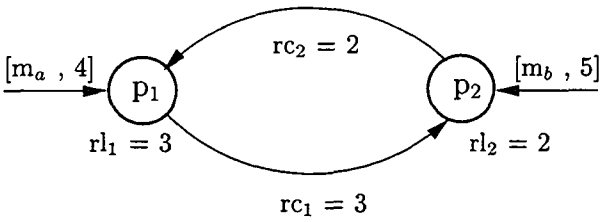

Figura 2.5: Situação de bloqueio.

Nesta situação, p_1 não pode consumir [m_a , 4] pois existe a possibilidade de receber uma mensagem com estampilha $2 \leq t < 4$ de p_2 e, por sua vez, p_2 não pode consumir [m_b , 5] pois pode receber uma mensagem estampilhada $3 \leq t < 5$ de p_1. Como a decisão de cada processo é local ao mesmo, sem informações externas, a chegada de novas mensagens não mudará a situação de bloqueio.

Duas estratégias podem ser utilizadas para tratar da possibilidade de bloqueios: através de mecanismos especiais, pode-se *prevenir* a ocorrência de bloqueios ou *detectar e resolver* eventuais bloqueios.

Prevenção de Bloqueios

Uma proposta para prevenir bloqueios utiliza mensagens de controle para evitar que tais situações ocorram. Essas mensagens são chamadas de "mensagens nulas" [CM79, Maz94]. Ao enviá-las, um processo comunica a outro uma "previsão" (ou *lookahead*) sobre seu comportamento futuro. Ao enviar uma mensagem nula [$null$, $rl_i + \delta$], com $\delta \geq 0$ e rl_i seu relógio local, o processo se compromete a não enviar novas mensagens antes de δ unidades de tempo simulado. Esta previsão é calculada a partir de dados como a duração mínima do tratamento de cada mensagem, o comportamento do processo, etc.

Neste método os processos mantêm seus relógios de canal atualizados através do envio de mensagens nulas. Convém ressaltar que essas mensagens nulas não correspondem à nenhuma atividade específica do sistema físico, mas servem apenas ao propósito de sincronização entre processos.

Toda vez que um processo envia uma mensagem não-nula por um canal de saída, ele envia outra, nula, através de cada um dos outros canais de saída. Esta mensagem

não contém nenhuma informação, apenas a previsão (δ) do próximo envio de mensagem do processo. A função dessa mensagem é fazer com que o relógio do canal de entrada do processo receptor (e o relógio do canal de saída do processor emissor) seja avançado. Tal mensagem indica que sobre aquele canal não será enviada outra mensagem com uma estampilha menor do que a estampilha da mensagem nula.

Ao receber uma mensagem nula, o processo receptor pode recalcular seu relógio de entrada (mínimo dos relógios dos canais de entrada) e eventualmente liberar para consumo mensagens cujas estampilhas sejam inferiores ao novo relógio de entrada. Ele pode também reavaliar seu relógio local, avançando-o, e com isso enviar novas mensagens nulas com melhores previsões δ a seus sucessores [Maz94].

Na figura 2.6, aparece um exemplo do uso de mensagens nulas. O processo p_1 envia uma mensagen nula a cada um de seus sucessores após o envio de uma mensagem não-nula $[m_a\ ,\ 7]$ para outro processo. Com o recebimento das mesmas, os processos p_2 e p_3 atualizam seus relógios de canal de entrada e chegam a um novo valor de relógio de entrada. Caso existissem mensagens que não pudessem ser consumidas pelos processos p_2 e p_3 antes daquele novo valor, as mesmas poderiam ser tratadas agora. p_2 e p_3 podem também enviar novas mensagens nulas a seus sucessores.

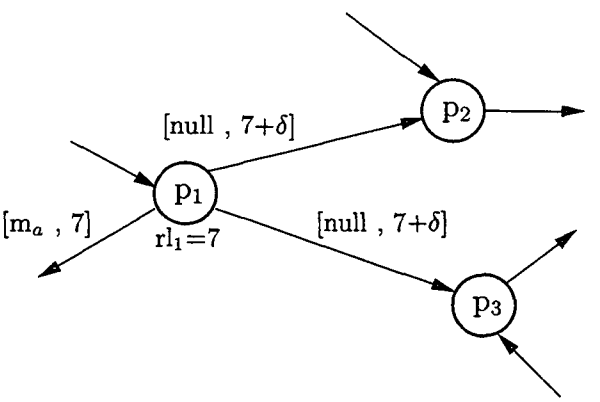

Figura 2.6: O uso de mensagens nulas.

Detecção e Resolução de Bloqueios

Outra forma de vencer os bloqueios é através da detecção e correção dessas situações [CM81, Mis86]. Para isso esta estratégia deixa a simulação prosseguir livremente até que seja detectado um bloqueio. A partir disso, tenta-se resolver o impasse para liberar a simulação.

Dois mecanismos diferentes são usados: um para detectar o bloqueio e outro para

solucioná-lo. O mecanismo para detecção pode ser encontrado em [Mis86], e baseia-se em técnicas clássicas de detecção de bloqueios [BT87, CJS87, SMP88, Ray92]. As estratégias de resolução se baseiam no fato de que a mensagem com a menor estampilha em toda a simulação é sempre segura para ser entregue ao processo receptor [Fuj90]. Com base nesse princípio a mensagem mais antiga em dado momento da simulação poderá ser entregue ao seu respectivo receptor. Para fazer isso deve-se descobrir qual a mensagem mais antiga na simulação inteira, solução que não se mostra muito eficaz.

A segunda solução é a proposta de execução em fases [CM81]. Chandy e Misra propõem o uso de um sítio especial no modelo, chamado controlador, que detecta o bloqueio e inicia a resolução do mesmo. A cada fase, o processo que faz parte do ciclo bloqueado envia a seus sucessores uma atualização hipotética, desconsiderando os seus canais vazios, da data mínima do próximo envio de mensagem [Maz94]. Com os valores recebidos de seus predecessores ele recalcula a sua própria data para a fase seguinte. Essa fase é repetida várias vezes até que se obtenham valores estáveis de relógio de canal. Neste ponto, pelo menos uma mensagem pode ser transmitida antes do próximo bloqueio [CM81].

Em [Fuj90] é demonstrado que essa estratégia aumenta a velocidade da simulação apenas em casos em que o bloqueio envolve muitos processos e o número de mensagens enviadas é muito grande. Existe um nível crítico de quantidade de mensagens. Quando a população de mensagens cai abaixo desse nível, a performance se torna pobre e relativamente constante.

2.6.2 Estratégias Otimistas

Nas estratégias otimistas todas as mensagens que já se encontram nos canais de entrada de um processo podem ser tratadas. Não se espera pela verificação de segurança das mesmas em relação à lei de causalidade. A simulação é executada normalmente até que uma mensagem com estampilha menor que as que já foram tratadas chegue ao processo. Esta situação pode ocorrer por força do assincronismo entre os diversos processadores envolvidos na simulação, e também pelo tempo (físico) de trânsito das mensagens entre eles. A chegada de uma mensagem retardatária provoca uma situação errônea que exige recuperação. Para isto a simulação deve de alguma forma tratar a mensagem que acabou de chegar antes das outras que já foram tratadas.

Uma abordagem do tipo otimista para simulação assíncrona distribuída foi proposta por Jefferson e Sowizral [Jef85, RW89, Fuj90]. Essa abordagem, chamada de *Time Warp*, utiliza o paradigma do "Tempo Virtual". No caso de uma violação de causalidade o tempo simulado pode ser retrocedido para se corrigir o erro.

O Retorno no Tempo Simulado

Quando uma mensagem retardatária chega a um processo (mensagem cuja data de envio é anterior ao relógio local do receptor), a simulação neste processo retrocede até um tempo simulado anterior ao da estampilha da mensagem recém-chegada.

A simulação no processo é então retomada, considerando a mensagem recém-chegada em sua posição correta. Retroceder a simulação em um processo implica em retornar a um estado anterior e eventualmente cancelar mensagens que tenham sido enviadas de modo errôneo, através do envio de pedidos de cancelamento (anti-mensagens). Para tal, cada processo deve armazenar seus estados anteriores, bem como os estados anteriores de suas entradas e as mensagens enviadas em suas saídas, o que pode constituir uma massa de dados considerável [Jef85].

TVG (Tempo Virtual Global)

Para armazenar todas as mensagens enviadas e os estados anteriores assumidos pelos processos é necessária uma quantidade significativa de memória, o que pode tornar inviável a simulação de grandes modelos.

Vimos nas estratégias pessimistas que em um dado instante a mensagem mais antiga em espera em toda a simulação pode ser tratada sem risco de violar a causalidade. Com base nessa constatação pode ser estabelecido um valor de tempo simulado, chamado "Tempo Virtual Global" (TVG), abaixo do qual os estados dos processos e os envios de mensagens são estáveis, e portanto podem ser descartados.

O TVG é calculado periodicamente levando-se em conta os relógios locais dos processos e as estampilhas das mensagens e anti-mensagens não tratadas ou em trânsito.

2.7 Conclusão

Este capítulo apresentou os principais tópicos referentes a simulação em ambientes distribuídos. Para fazer isso foram apresentados os dois princípios que regem todo sistema físico, e que deverão ser respeitados ao longo de toda simulação: o determinismo e a causalidade. Foram mostrados também os dois tipos de modelos de simulação existentes: o contínuo e o discreto. Dentro da abordagem discreta vimos as formas de converter os sistemas em estudo em modelos de simulação. Vimos que podemos fazer isso de uma forma orientada a eventos, ou orientada a processos.

É necessário controlar como o tempo simulado é avançado dentro de uma simulação. Para isto vimos que existem duas formas de condução. Podemos controlar uma simulação

através do tempo ou dos eventos do sistema. Na abordagem coordenada pelos eventos existe um mecanismo normalmente utilizado para garantir o respeito à causalidade: o escalonador.

Com essa base pronta pudemos expandir nosso estudo em direção ao nosso objetivo: a simulação ocorrendo em ambientes distribuídos. Foi apresentado o modelo que adotamos no nosso trabalho em termos de processos comunicantes através de troca de mensagens. Esse modelo servirá de base aos demais capítulos.

Vimos que o maior problema nesse tipo de simulação é a garantia do respeito à causalidade, já que na ausência de um relógio único devemos nos preocupar com o sincronismo entre os processos. Na busca de soluções vimos quais as abordagens existentes: a simulação síncrona e a assíncrona.

Dentro da simulação assíncrona mostramos a existência de duas abordagens diferentes para tentar garantir a causalidade: as estratégias otimistas e pessimistas. Vimos sucintamente como o método otimista funciona e quais os pontos principais nessa abordagem. Na abordagem pessimista vimos quais os tópicos de maior importância para se manter a coerência dentro de uma simulação. Vimos como tentar resolver o problema do bloqueio através da prevenção da ocorrência do mesmo mantendo os relógios dos canais sempre atualizados.

No capítulo a seguir veremos como uma simulação pode ser implementada através de linguagens orientadas a objetos. Veremos quais a formas adotadas para isso, e qual escolhemos para o nosso trabalho. Será então mostrado como a simulação pode ser desenvolvida através dessa escolha.

Capítulo 3

Simulação Orientada a Objetos

3.1 Introdução

No capítulo anterior estudamos os conceitos fundamentais da simulação por computador, seus mecanismos básicos e a implementação destes. Abordamos também os principais problemas associados à simulação distribuída e algumas soluções propostas na literatura. Neste capítulo devemos considerar a simulação distribuída ocorrendo em uma plataforma orientada a objetos.

É inegável e bem conhecido o conjunto de facilidades que obtemos com a programação orientada a objetos: o encapsulamento, o reaproveitamento de código, o polimorfismo entre outras. A aceitação da programação orientada a objetos é tão grande hoje em dia, que já existem ambientes de desenvolvimento para praticamente todos os sistemas operacionais.

O uso de objetos em simulações a eventos discretos não é recente, muito pelo contrário. Esse paradigma tem suas próprias origens na necessidade de modelar detalhadamente entidades do mundo real para fins de simulação. Prova disto é a linguagem *Simula* [BDMN73], a primeira a incorporar os conceitos de classes, objetos, métodos e atributos.

No decorrer dos últimos anos a orientação a objetos se tornou uma poderosa ferramenta de programação geral, trazendo benefícios também para o contexto da simulação.

3.2 Simulando com Objetos

O uso da programação orientada a objetos na construção de simulações a eventos discretos é bastante simples e intuitivo. Nesse contexto, entidades reais são modeladas por objetos, que pertencem a determinadas classes, de acordo com suas características e funcionalidades. Interações entre entidades são vistas como ativações de métodos entre objetos.

Assim como no sistema real existem entidades ativas (como robôs, carrinhos, etc.) e passivas (como depósitos, *buffers*, etc.), no ambiente de simulação devem existir objetos ativos e passivos. A possibilidade de existência de mais de um objeto ativo na simulação (o que é a situação normal) leva à necessidade de suporte à concorrência entre objetos no ambiente de simulação. Desta forma, ao contrário dos ambientes genéricos de programação a objetos, o ambiente de programação de simulações orientadas a objetos deve prover facilidades para a gestão da concorrência entre objetos, como suporte a *threads*, semáforos, etc.

Da mesma forma que em uma simulação a eventos discretos clássica, em um ambiente seqüencial de simulação orientado a objetos, estes também estão sob o controle de um escalonador. Esse mecanismo se encarrega de ativar os objetos de acordo com a evolução do tempo simulado, de modo a preservar o princípio de causalidade. Em um dado instante, o objeto em execução pode solicitar a execução ou suspensão de outros objetos ativos, manipular objetos passivos e finalmente suspender-se à espera de uma reativação futura.

Normalmente encontramos duas abordagens para construção de ambientes de simulação orientados a objetos:

- *Linguagens específicas.* Nesta abordagem é desenvolvida uma nova linguagem de programação, estendendo-se uma já existente ou criando-se uma inédita. Esta abordagem tem a vantagem de que o compilador ou o pré-processador pode prover verificação estrita de tipos, geração de código otimizado, etc. [MB95]. Como exemplo dessa abordagem podemos citar a linguagen Simula [BDMN73].

- *Bibliotecas especializadas.* Nesta abordagem constrói-se uma biblioteca de funções específicas à simulação, a partir de uma linguagem padronizada e bem conhecida. As principais vantagens deste modelo são que o programador não precisa mudar o seu ambiente de programação, e, em princípio, uma maior variedade de plataformas físicas de execução pode ser considerada. Como exemplos dessa abordagem podemos citar as bibliotecas C++SIM [Uni94] e CNCL [JSB+96], ambas baseadas em C++.

Na seqüência deste texto vamos apresentar com mais detalhes C++SIM, uma típica biblioteca para a programação de simulações a eventos discretos orientadas a objetos. C++SIM é uma biblioteca simples, mas poderosa e flexível, e será de grande importância para o desenvolvimento do nosso trabalho, como veremos no capítulo 5.

3.2.1 C++SIM

A bibilioteca C++SIM [Uni94] foi construída em C++, no contexto do projeto Arjuna [P+94]. Entre suas características mais interessantes podemos ressaltar:

- *Facilidade de aprendizado e uso.* As classes são simples e de fácil entendimento.
- *Correta abstração.* Como a mesma é construída em C++ não existe diferença entre o paradigma usado no modelo a ser simulado e o da biblioteca.
- *Flexibilidade e extensibilidade.* É possível acrescentar novas classes, além de modificar as já existentes.
- *Eficiência.* Como é gerado em C++, o código final é rápido e de pequeno tamanho.
- *Portabilidade.* O código fonte da biblioteca é disponível e pode ser facilmente compilado para uma variedade de sistemas operacionais.

A grande vantagem dessa biblioteca é a construção do ambiente de simulação baseada em uma hierarquia de classes. Com isso pode-se modificar qualquer uma das suas propriedades com a alteração de poucos itens.

A biblioteca C++SIM se baseia nos conceitos apresentados pela linguagem Simula [BDMN73]. A simulação é formada por objetos ativos, que são instâncias de classes em C++. Cada objeto ativo no modelo possui uma *thread* associada à sua execução, o que permite usar o conceito de *multithreads* com certa transparência.

Na modelagem de um sistema usando a biblioteca C++SIM, esta fornece as classes básicas para a modelagem, e o usuário precisa então definir as classes que modelem o sistema a ser simulado. As classes básicas contém os métodos principais para controlar todo o processo de simulação. Para construir o modelo a ser simulado o usuário deve então definir, a partir das classes básicas, novas classes que irão compor a imagem do modelo real. A figura 3.1 na página 20 mostra um esquema explicativo do processo de modelagem.

A simulação será executada pelo programa resultante da compilação das classes básicas com as definidas pelo usuário. Assim, a classe `main` resultante dessa compilação irá iniciar o processo de simulação, através da instanciação dos vários objetos que formam o modelo a ser simulado. Durante a simulação, objetos poderão ser criados ou destruídos, dependendo do modelo em execução. Ao final da simulação todos os objetos da simulação são destruídos e o controle é retornado à classe `main`.

A biblioteca C++SIM é composta por um conjunto reduzido de classes, cujos elementos principais apresentaremos na seqüência do texto. Para ajudar na compreensão a figura 3.2, na página 21, mostra um esquema com a relação de herança entre as principais classes da biblioteca.

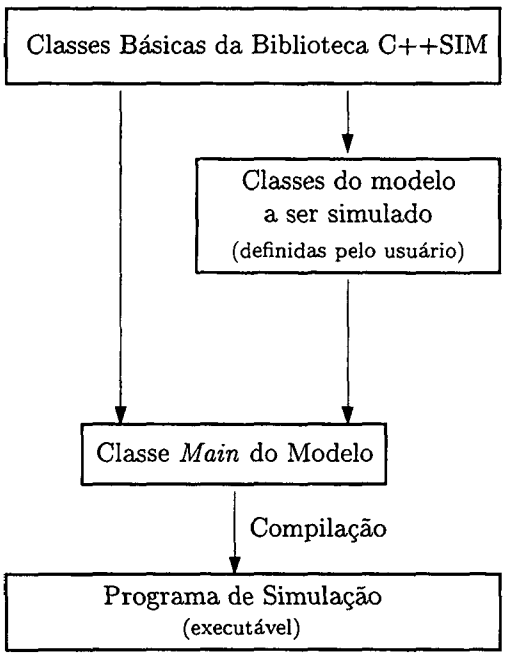

Figura 3.1: Modelagem de processos

Processos - Classe *Process*

A classe *Process* é a responsável pela definição de todas as entidades participantes da simulação, aqui também chamadas de "processos". Nesta biblioteca, cada entidade ativa possui uma *thread* independente associada ao mesmo. Isso proporciona a noção de atividade necessária para a divisão da simulação em processos[1]. A classe *Process* define os estados possíveis que um processo pode assumir a qualquer instante da simulação [Uni94]:

- *Ativo.* É o processo que está sendo executado no momento. Este, antes da sua execução, estava na "cabeça" da fila do escalonador.
- *Suspenso.* É o processo que está na fila de eventos do escalonador. Este se tornará ativo no tempo de ativação correspondente constante na fila.
- *Passivo.* É o processo que não está na fila de eventos. Este não será executado a menos que outro processo o coloque de volta na fila.
- *Terminado.* É o processo que não está na fila de eventos e nem possui mais ações a serem executadas. Este não poderá mais ser ativado na simulação atual.

[1]No entanto uma simulação em C++SIM é constitutída por um único processo UNIX. A divisão em *threads* se dá no interior desse processo.

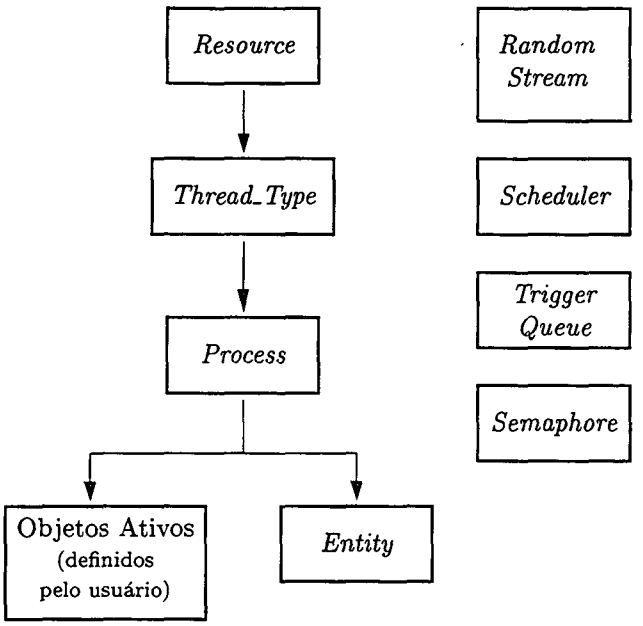

Figura 3.2: As classes de C++SIM

São também definidos outros métodos para controlar o estado de um processo: verificação da situação do processo (terminado ou passivo); reescalonamento do processo (na fila de eventos); cancelamento do processo; e ativação (ou reativação) de um processo em qualquer posição do tempo simulado.

Escalonador - Classe *Scheduler*

A biblioteca C++SIM utiliza o mesmo paradigma apresentado no capítulo 2 para controlar sua lista de eventos, ou seja, sob uma simulação a eventos discretos o escalonador controla qual será o próximo evento a se tornar ativo. Neste caso, os eventos correspondem às futuras reativações de processos suspensos. A partir do momento em que um processo é reativado (executado), ele pode gerar outros eventos (pedidos de reativação de processos) que serão enviados para o escalonador e inseridos levando-se em conta a data de ativação dos outros eventos já presentes. Quando não houver mais eventos na lista, ou seja, a fila de processos do escalonador estiver vazia, então a simulação será concluída.

Funções Aleatórias - Classe *Random Stream*

Para a construção de um modelo de simulação é normalmente necessário o uso de geradores de números aleatórios para obtermos um comportamento próximo da situação real.

A biblioteca C++SIM faz uso de algoritmos numéricos para construir uma classe básica

de distribuição aleatória uniforme entre 0 e 1 (*RandomStream*), a partir da qual outras classes são definidas: *UniformStream*, *ExponentialStream*, *NormalStream*, etc.

Entidades Assíncronas - Classe *Entity*

Em muitas circunstâncias a execução de um evento pode estar condicionada não somente a uma data de ocorrência, como vimos até agora, mas também a outras condições, como o término da execução de outro evento, a liberação de algum recurso (semáforo), etc. Eventos dessa ordem podem ocorrer a qualquer momento no tempo simulado, e por isso são chamados "eventos assíncronos".

O mecanismo básico do escalonador, como visto até agora, não tem flexibilidade suficiente para tratar de forma eficiente eventos dessa natureza. Por esta razão, a biblioteca C++SIM estendeu a noção de processos de simulação, inerentemente síncronos, para *entidades de simulação*, cujo escalonamento pode depender de condições assíncronas. As entidades são tratadas pelo escalonador como processos. Além dos estados possíveis aos processos, as entidades podem assumir outros dois estados:

- *Esperando.* O processo está aguardando a ocorrência de um evento específico para ser executado, que pode ser uma data determinada, o término da execução de outro processo, a liberação de um semáforo, ou outra condição especificada pelo usuário. O processo em espera não é colocado na fila do escalonador, mas em filas especiais chamadas "*trigger queues*". Todos os processos que esperam a ocorrência do mesmo evento são agrupados em uma mesma *trigger queue*. Na ocorrência do evento a aplicação pode então optar por só "disparar" o primeiro processo da fila ou todos os processos que estão na espera daquele evento.

- *Interrompido.* O processo que outrora estava em estado de espera foi interrompido antes que o evento pelo qual aguardava ocorresse.

Semáforos - Classe *Semaphore*

Para que um processo ou entidade consiga alocar com exclusividade algum recurso disponível na simulação, a biblioteca C++SIM coloca semáforos à disposição do modelo. Quando um processo tenta alocar um recurso tentando pegar um semáforo, e o mesmo se encontra indisponível, esse processo é suspenso automaticamente e colocado na *trigger queue* correspondente ao semáforo, até que este seja disponibilizado. Para tornar um semáforo disponível outra vez o processo que o alocou deve chamar um método para liberá-lo.

3.2.2 Uma Simulação em C++SIM

Inicialmente todas as instâncias da classe *Process* se encontram no estado "passivo", e devem ser ativadas para tomar parte na simulação. Essa ativação é responsabilidade do primeiro objeto "processo" para o qual o controle é passado logo após a inicialização da simulação. Normalmente a *thread* principal do programa de simulação cria um processo controlador que é responsável por coordenar a execução da simulação inteira. Esse controlador cria e ativa todos os processos da simulação, bem como o escalonador. O mesmo é também responsável pela definição de métodos para a suspensão da *thread* principal, permitindo a execução e o encerramento da simulação.

Normalmente o controlador deve possuir pelo menos dois métodos:

- *Await.* Esse método é chamado dentro da *thread* principal, suspendendo-a e transferindo o controle para o processo controlador.
- *Exit.* Esse método é chamado para o encerramento da simulação.

Para o encerramento da simulação deve-se suspender os processos que ainda estão executando e retornar o controle para o processo controlador. Por sua vez, este limpa o escalonador e suspende a execução deste. Feito isso, o processo controlador passa os processos que estão em estado "suspenso" para o estado "terminado" e destrói as instâncias desses objetos na memória. Após isso o controle é passado para a *thread* principal para que esta termine o programa normalmente.

Existem métodos especiais para encerrar a simulação a qualquer instante. Esses métodos são úteis quando existe uma situação especial que possa causar o encerramento da execução da simulação. Além disso, existem ainda classes extras para o controle estatístico e para a depuração dos modelos.

3.3 Simulando com Objetos Distribuídos

Além dos mecanismos básicos necessários à gestão de simulações a eventos discretos, a construção de um ambiente de simulação orientada a objetos sobre um contexto distribuído possivelmente heterogêneo implica em:

- Mapear modelos expressos em termos de objetos e métodos a uma estrutura mais apropriada à execução em um contexto distribuído, como processos e mensagens.
- Distribuir os objetos entre as diversas máquinas de acordo com políticas de mapeamento específicas, visando a minimização das comunicações entre máquinas, o equilíbrio das cargas de trabalho, etc.

- Prover meios para a comunicação entre objetos em máquinas distintas, preservando a semântica "ativação de método" e, se possível, abstraindo a separação física entre os objetos e as diferenças entre máquinas e sistemas operacionais envolvidos.
- Sincronizar a execução dos objetos, de maneira a fazer evoluir o tempo simulado sem violações do princípio de causalidade.

Existem diversas propostas de ambientes orientados a objetos para a simulação distribuída, alguns deles baseados em linguagens dedicadas, como MOOSE [WB94], Sim++ [LB91] e outras baseadas em bibliotecas de uma linguagem convencional, como COMPOSE [MB95] e PROSIT [MM94]. Nas seções a seguir vamos apresentar com mais detalhes duas dessas propostas, que nos pareceram as mais interessantes: MOOSE e PROSIT.

3.3.1 MOOSE

A linguagem *MOOSE* [WB94] é uma linguagem de simulação orientada a objetos baseada em *Maisie* [Bag91], que por sua vez é baseada em C. Ambas as linguagens Maisie e MOOSE permitem a construção incremental de modelos de simulação a eventos discretos. Em um primeiro passo o usuário constrói modelos para simulação seqüencial, que em seguida podem ser refinados e otimizados para execução paralela segundo um determinado esquema de sincronização, pessimista ou otimista.

Um programa MOOSE define uma coleção de entidades que se comunicam através de trocas de mensagens. Cada entidade define um objeto, ou seja, um conjunto de estruturas de dados e de métodos que operam sobre esses dados. Aos métodos podem ser opcionalmente associadas pré-condições sobre seus parâmetros e os dados locais da entidade, para controlar a execução do método. Assim, um método só será executado se sua pré-condição associada for verdadeira naquele instante. As pré-condições associadas aos métodos podem ser herdadas por novas entidades, simplificando assim a criação incremental de modelos.

O mecanismo de herança pode ser empregado para a criação de novas entidades a partir das já existentes (MOOSE não suporta múltipla herança). Esse mecanismo é extensivamente usado durante a fase de refinamento do modelo, na qual as entidades do modelo já construído para a simulação seqüencial herdam características de sincronização distribuida de classes pré-definidas para estratégias pessimistas ou otimistas. Em cada caso será necessário redefinir métodos para atualização de previsões (*lookaheads*) (nas abordagens pessimistas) ou para o armazenamento e recuperação de estados (nas abordagens otimistas).

Implementações de MOOSE estão atualmente disponíveis para arquiteturas seqüenciais ou paralelas. As implementações seqüenciais podem ser executadas em qualquer plataforma possuindo um compilador C. As implementações paralelas somente podem ser executadas em arquiteturas que suportem o ambiente de programação distribuida *Cosmic* [SSS88] ou SUN-IPC.

3.3.2 PROSIT

O ambiente de desenvolvimento PROSIT [MM94, FMSM94], proposto no INRIA (França) é baseado em uma extensão da linguagem orientada a objetos *Sather*, desenvolvida no *Computer Science Institute*, Berkeley. A linguagem *Sather* provê facilidades para a expressão e uso eficiente do paralelismo em um contexto orientado a objetos, como chamadas a métodos remotos, migração de objetos, memória compartilhada virtual, etc. Além disso, ela suporta o conceito de *threads* e provê mecanismos para sincronização entre *threads*.

O ambiente PROSIT define um conjunto de classes de simulação e de modelagem (as classes básicas). As primeiras se referem à fase de controle da simulação e as segundas são usadas para construir o modelo.

Com o objetivo de esconder os mecanismos de simulação ao usuário final do simulador, um programador define um conjunto de classes (herdeiras das classes básicas) para um campo específico de aplicação. Essas classes adicionais permitem que o usuário final construa um modelo em um nível de descrição acima daquele correspondente ao paradigma de simulação. Assim, o usuário não precisará saber como ocorre a simulação para construir o seu modelo. No final o modelo será constituído da classe *main*, que por sua vez é feita de instâncias de classes definidas pelo usuário e de classes básicas do sistema.

No ambiente PROSIT, o simulador é organizado através da cooperação de vários subsimuladores e de um supervisor de simulação (figura 3.3). Cada subsimulador tem um núcleo (*kernel*) de simulação seqüencial que processa um grupo de entidades do modelo. Subsimuladores interagem entre si para implementar um método de sincronização (*Time-Warp*, mensagens nulas, etc.). O supervisor tem a função de carregar o modelo e de gerenciar a interface com o usuário durante a simulação. Ele também é responsável por serviços de comunicação global e por outros serviços essenciais (*tracing*, equilíbrio de cargas, etc.)

A simulação em um subsimulador funciona da mesma forma que uma simulação seqüencial. O núcleo da simulação tira o primeiro evento de uma lista de eventos (menor tempo de ativação) e executa o método do cliente (referenciado pelo evento) até que ele consiga de volta o controle e possa processar o próximo evento da lista. O controle retorna ao núcleo de simulação tão logo o método de controle de tempo seja chamado pelo cliente

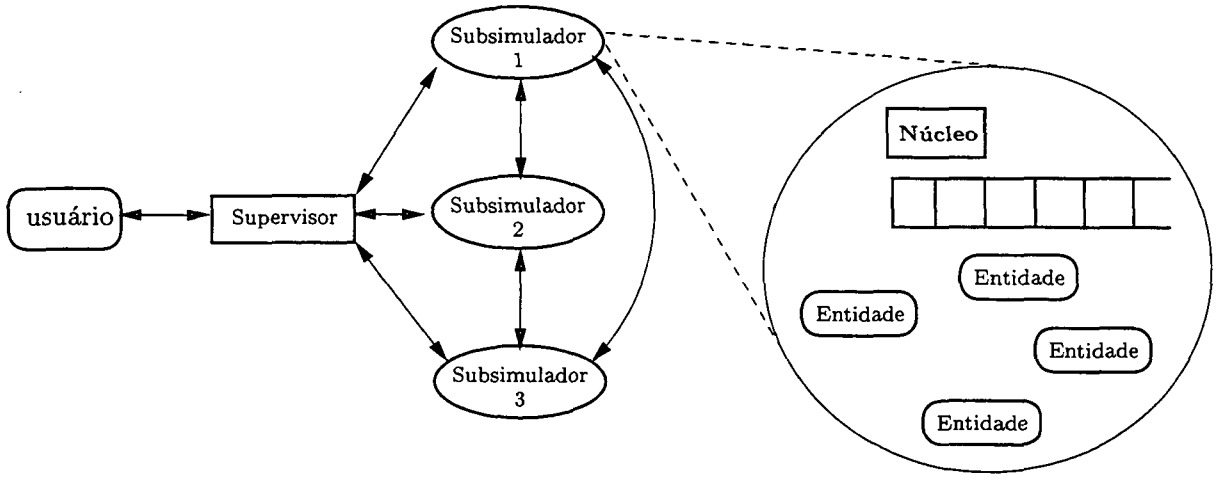

Figura 3.3: Arquitetura do ambiente PROSIT

ativo.

As comunicações de baixo nível no ambiente PROSIT são manipuladas através da biblioteca de comunicação PVM (*Parallel Virtual Machine*) [Sun90]. Conseqüentemente, PROSIT é capaz de executar em sistemas distribuídos que suportem PVM, incluindo grupos de estações de trabalho heterogêneas conectadas através de LAN e sistemas MIMD. Contudo, a arquitetura de referência é a máquina *Meiko Computing Surface.*

3.4 Conclusão

Neste capítulo vimos que o uso do paradigma de orientação a objetos em simulações a eventos discretos não é uma idéia recente, e que esse paradigma tem suas origens no próprio contexto da simulação. A abordagem por objetos traz benefícios claros ao processo de modelagem e à própria construção da simulação.

Apresentamos em detalhe C++SIM, uma biblioteca escrita em C++ para a construção de simulações a eventos discretos orientadas a objetos. Apesar de simples, essa biblioteca mostrou-se uma ferramenta poderosa e versátil na construção de simulações seqüenciais.

Abordamos também os principais problemas encontrados na execução de simulações orientadas a objetos sobre uma plataforma distribuída, e apresentamos duas propostas de sistemas que buscam construir ambientes para a execução distribuída de simulações orientadas a objetos.

Um problema importante observado nos ambientes apresentados (e em outros estudados) diz respeito à possível heterogeneidade do suporte de execução. Por exemplo,

a proposta PROSIT incorpora no ambiente de simulação funcionalidades para tentar adaptar-se a um suporte heterogêneo, mas essa abordagem, vai de encontro à tendência atual de se "construir" a homogeneidade a nível do sistema operacional, para oferecer às aplicações uma visão homogênea e coerente do suporte de execução. Exemplo disso é a proposta CORBA (*Common Object Request Broker Architecture*), voltada às aplicações orientada a objetos, que estudaremos em detalhe no próximo capítulo.

Nosso trabalho visa a definição de um ambiente de simulação a eventos discretos orientada a objetos sobre uma plataforma aberta. Os sistemas atualmente existentes incorporam seus próprios meios para a resolução (limitada) da heterogeneidade, quando não a ignoram. Desta forma, optamos por empregar uma plataforma de execução aberta nos moldes da norma CORBA, como veremos no próximo capítulo, e sobre ela construir um ambiente de simulação orientado a objetos.

Para as funcionalidades básicas da simulação a eventos discretos utilizaremos a biblioteca C++SIM, cuja simplicidade e versatilidade se mostrou adequada a esse propósito. No entanto, cabe lembrar que a biblioteca C++SIM foi desenvolvida visando a implementação de simulações seqüenciais, não dispondo portanto de mecanismos para a comunicação e sincronização de objetos executando em um ambiente distribuído. Nosso trabalho consistirá portanto no uso dessa biblioteca como elemento de base na construção de um ambiente de programação de simulações distribuídas orientadas a objetos. No próximo capítulo estudaremos a plataforma CORBA, capaz de nos prover mecanismos poderosos para a comunicação transparente entre objetos em um ambiente heterogêneo.

Capítulo 4

A Arquitetura Aberta CORBA

4.1 Introdução

Nosso objetivo é estudar a melhor forma de viabilizar simulações de modelos de grandes dimensões. No caminho para esse objetivo já vimos as vantagens de se utilizar computação distribuída. Também apresentamos o uso vantajoso da orientação a objetos.

Para a realização desse objetivo também é conveniente o uso de uma plataforma aberta de comunicação, pois com a grande diversidade de sistemas existentes no mercado é condição importantíssima o uso completo dos recursos disponíveis. O ideal para o nosso caso seria o uso de uma plataforma aberta e orientada a objetos. Isso já está disponível na arquitetura CORBA (*Common Object Request Broker Architecture*).

A arquitetura CORBA foi proposta pela OMG (*Object Manager Group, Inc.*). Esta organização internacional formada por mais de 500 membros, entre usuários e empresas desenvolvedoras de *software*, foi fundada em 1989 com o objetivo de promover a teoria e prática da tecnologia orientada a objetos no desenvolvimento de *softwares*. A mesma procura estabelecer as linhas que servirão de guia para a indústria, bem como as especificações de gerenciamento de objetos para prover uma estrutura comum para desenvolvimento de aplicações. Os objetivos primários desta organização são a reusabilidade, a portabilidade, e a interoperabilidade de *softwares* orientados a objetos em ambientes distribuídos heterogêneos [Obj95].

4.2 A Arquitetura OMA

Para conseguir atingir seus objetivos, a OMG especificou uma arquitetura que serve como base para a cooperação entre aplicações constituídas por objetos distribuídos [Siq95]. Tal arquitetura é conhecida como OMA (*Object Management Archicture*). Esse modelo é

mostrado na figura 4.1.

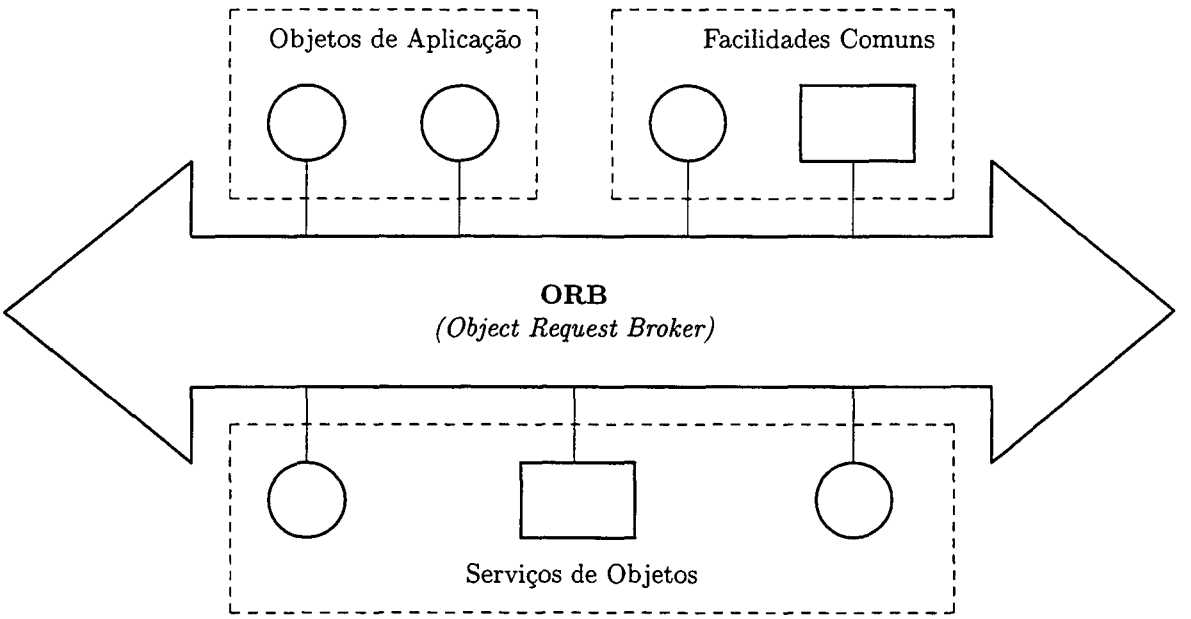

Figura 4.1: Modelo de referência da OMA

No desenho podemos notar:

- Objetos de aplicação (*Application Objects*). São os produtos feitos pelo usuário. Cada qual visando atender a um objetivo específico.
- Facilidades comuns (*Common Facilities*). Constituem uma coleção de serviços que podem ser compartilhados pelas mais diversas aplicações.
- ORB (*Object Request Broker*). Parte fundamental para a criação de aplicações feitas a partir de objetos distribuídos e para a interoperabilidade entre aplicações que rodem em ambientes heterogêneos. Este permite que objetos façam e recebam, transparentemente, requisições e respostas em ambientes distribuídos.
- Serviços de objetos (*Object Services*). É uma coleção de serviços (interfaces e objetos) que fornecem funções básicas para o uso e implementação de objetos.

A parte mais importante do modelo é o ORB. O mesmo pode ser considerado o núcleo do modelo de referência.

O CORBA é uma especificação das interfaces e arquitetura do ORB.

4.3 A Estrutura de um ORB

A arquitetura CORBA é estruturada de forma a permitir a integração de uma grande variedade de sistemas de objetos.

A estrutura de um ORB é melhor compreeendida através da figura 4.2. Nela podemos ver a situação em que uma requisição está sendo enviada por um cliente para uma implementação de objeto.

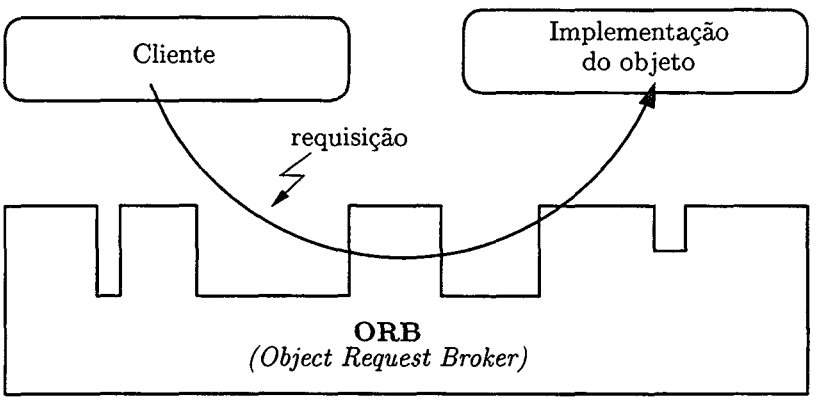

Figura 4.2: Uma requisição enviada através do ORB

O ORB é responsável pela busca do objeto a que se destina a requisição feita pelo cliente. É responsável também pela preparação da implementação do objeto para a recepção da requisição, além da entrega da mesma, de uma forma que o objeto a aceite. Cabe ao ORB também o retorno dos parâmetros de saída da requisição, caso haja algum, que serão entregues ao cliente.

A grande vantagem aqui é que o cliente tem contato com uma interface que é completamente independente da localização do objeto de interesse. Também não importa qual linguagem de programação foi utilizada na sua implementação, ou de quaisquer outros aspectos relacionados à diversidade encontrada no ambiente distribuído em que se encontra.

Podemos ver na figura 4.3 a estrutura de um ORB e suas interfaces, pelo meio das quais acontece a interação com os clientes e implementações de objetos.

As definições de interfaces de objetos podem ser feitas de duas maneiras diferentes. As interfaces podem ser definidas estaticamente em uma linguagem de definição de interface (*Interface Definition Language* - IDL), que define os tipos de objeto de acordo com as operações que podem ser executadas pelos mesmos e pelos parâmetros dessas operações; ou as interfaces podem ser adicionadas a um serviço de repositório de interfaces. Esse serviço representa os componentes de uma interface como objetos, para posterior acesso.

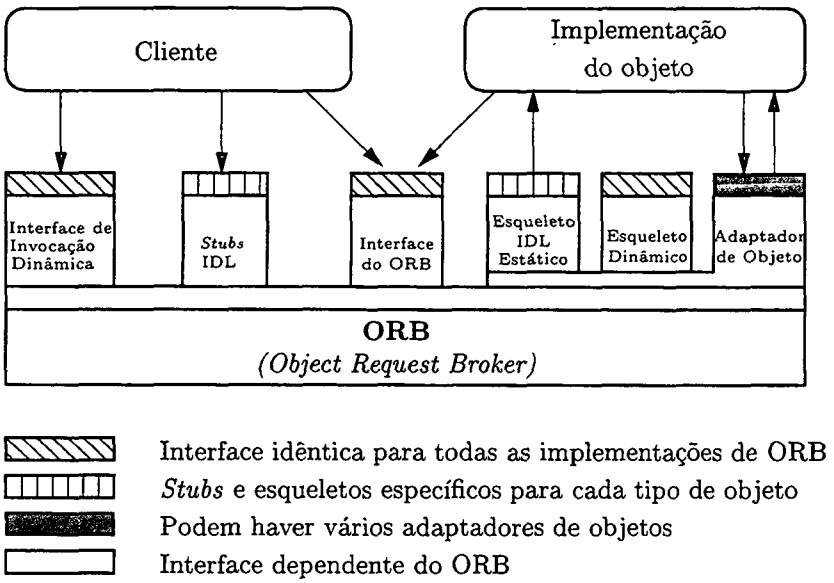

Figura 4.3: A estrutura de um ORB

Para fazer uma requisição de um serviço a um objeto cada cliente pode fazer uso de um de três meios. O primeiro através das *stubs* geradas na compilação da descrição de interface (arquivo IDL) do objeto destinatário do pedido, no caso de ter acesso às mesmas. O segundo é através da utilização da interface de invocação dinâmica (*Dynamic Invocation Interface* - DII). Para que isto seja possível, a interface do objeto deve ter sido adicionada previamente ao repositório de interfaces, permitindo acesso *runtime* a esses componentes através da interface. A vantagem deste último é que se tem a mesma interface independente da interface do objeto alvo [Obj95]. E através do terceiro meio é possível ao cliente interagir diretamente com o ORB fazendo uso de algumas funções específicas.

Para um cliente executar uma requisição, ele deve conhecer a referência do objeto, o tipo do objeto e a operação que deseja executar. Com esses dados o cliente pode chamar a rotina *stub* correspondente, ou construir a requisição dinamicamente através da DII.

Executada a requisição, o ORB localiza o código da implementação apropriado, transmite os parâmetros e transfere o controle para a implementação de objeto através de um esqueleto IDL ou um esqueleto dinâmico. Os esqueletos são específicos à interface e ao adaptador de objeto.

Na execução do serviço solicitado, a implementação do objeto pode acessar alguns serviços fornecidos pelo ORB através de um adaptador de objeto. Podem existir vários adaptadores de objetos, cabendo a implementação do objeto escolher qual irá usar, com base no tipo de serviço que deseja do ORB.

Quando a requisição é completada, o controle e os valores de saída são retornados ao cliente.

O ORB provê mecanismos de representação de objetos e de comunicação para que seja possível processar e executar as requisições. O CORBA foi estruturado sobrepondo seus componentes ao núcleo ORB, permitindo que as diferenças entre os núcleos sejam mascaradas completamente.

4.4 CHORUS/COOL

Para este trabalho utilizamos o *software* CHORUS/COOL-ORB [Cho96a, Cho96b], que é um ambiente de desenvolvimento combinado, baseado no padrão CORBA 2.0 da OMG. CHORUS/COOL-ORB provê interfaces de programação IDL sobre uma variedade grande de sistemas heterogêneos (AIX, SunOS, Solaris, Windows 95 e NT, Linux, SCO OpenServer 5, etc.). Ele é baseado na linguagem C++ e possui um compilador que converte CORBA/IDL em C++.

O CHORUS/COOL-ORB nos pareceu adequado ao desenvolvimento de nossa aplicação por possuir várias qualidades que nos pareceram úteis:

- Ambiente de execução para objetos (*cápsulas*) e sua ativação (*atividade*).
- Invocação transparente de objetos remotos.
- Sincronização usando *mutexes*, semáforos e travas (*locks*) concorrentes.
- Notificação para eventos desconhecidos (*foreign*).
- Suporte gráfico X11 para aplicações UNIX.
- Comunicação otimizada pelo *runtime* do CHORUS/COOL-ORB quando ambos, cliente e servidor, estão no mesmo espaço de endereço.
- Execução *multi-thread* no sistema operacional utilizado no nosso caso (Solaris 2.5, usando compilador Sparc C++ V.4.1).
- Disponibilidade para várias plataformas (Linux, Solaris, Windows 95, SunOS, etc.).

4.4.1 Ferramentas de Desenvolvimento

A principal ferramenta de desenvolvimento no ambiente COOL-ORB é o "CHIC" (*CHORUS IDL Compiler*), o compilador que traduz as definições IDL para C++ (figura 4.4).

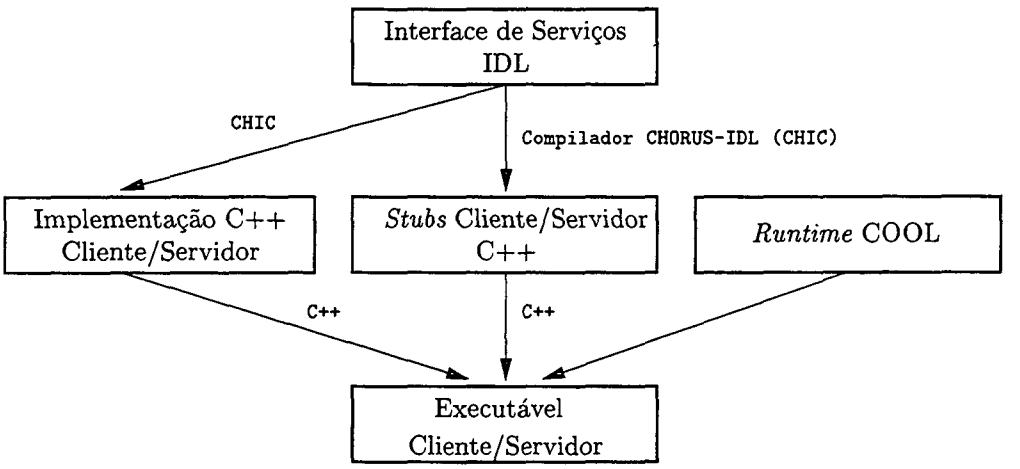

Figura 4.4: O compilador CHIC

A partir do arquivo definido em IDL, o compilador CHIC gera o código C++ usado pelo cliente para fazer uma requisição (*stub*), e pelo servidor para processar uma requisição (esqueleto). O código gerado é organizado em:

- Declaração de tipos e *stubs* em C++ usados para desenvolver o cliente.
- Implementação C++ dos *stubs*.
- Definição de tipos e esqueletos em C++ usados para desenvolver o servidor.
- Implementação C++ do esqueleto.

O código gerado é compilado usando as ferramentas C++ instaladas na plataforma de desenvolvimento.

Para aumentar a portabilidade, COOL-ORB inclui também um conjunto de facilidades de compilação baseado no *imake* (*software* básico do sistema operacional UNIX).

4.4.2 Suporte *Runtime*

Serviços adicionais definidos em IDL são fornecidos:

- Serviço de nomeação (*Naming Service*). Provê nomes simbólicos para referência de objetos, quando da invocação por parte de um cliente.
- Serviço de nodos (*Node Service*). O mesmo é uma facilidade administrativa COOL-ORB rodando em cada nodo, onde uma aplicação é executada. Ele provê acesso ao nodo para a criação de cápsulas e objetos de sincronização distribuídos.

- Serviço de sincronização (*Synchronisation Service*). Permite o gerenciamento de objetos de sincronização distribuídos básicos como semáforos, *mutexes*, e objetos de trava (*lock*).

- Serviço de domínio (*Domain Service*). Representa os pontos de conexão para os nodos (que são parte do domínio), e para os domínios externos.

- Serviço de Grupos (*Group Service*). Permite a um cliente requisitar uma operação em um ou mais servidores de uma maneira transparente. O serviço de grupos fornece suporte para o gerenciamento de grupos, incluindo inserção dinâmica ou retirada de servidores em um grupo.

4.5 Conclusão

Neste capítulo vimos a arquitetura CORBA da OMG. Conhecemos como funciona esta arquitetura aberta para comunicação entre plataformas distintas.

O CORBA é baseado no paradigma de orientação a objetos. Através de uma arquitetura que pode ser implementanda em quase todos os sistemas operacionais e em várias linguagens diferentes, é possível executar qualquer *software* que use suas funções padronizadas, permitindo que o mesmo possa se comunicar com outros objetos através de uma rede baseada em TCP/IP.

Foi apresentada a arquitetura OMA que serve de base para tal comunicação. Conhecemos sucintamente alguns de seus componentes e vimos qual o função dos mesmos na arquitetura OMA.

Em seguida conhecemos a estrutura ORB. Vimos como a comunicação entre os objetos se faz. Para permitir essa interatividade é necessária uma definição de interfaces feitas astravés dos arquivos IDL. Vimos que as comunicações podem ser feitas através de três formas distintas: através dos *stubs*, da interface de invocação dinâmica (DII), ou através de acesso direto à funções específicas do ORB.

Foi apresentado também o *software* CHORUS/COOL. Este ORB é o que usaremos neste trabalho para desenvolver a simulação distribuída. Conhecemos algumas das suas características e vimos como se desenvolve um *software* através de suas funções. E por último vimos algumas características adicionais definidas através dos arquivos IDL.

No capítulo seguinte será apresentada a nossa proposta para se conseguir executar uma simulação distribuída orientada a objetos, usando o CHORUS/COOL como meio de comunicação.

Capítulo 5

O Suporte de Simulação Proposto

5.1 Introdução

Neste capítulo apresentaremos nossa proposta de um suporte para a execução distribuída de simulações orientadas a objetos. Como principais preocupações que nortearam nosso trabalho podemos citar:

- A independência em relação à plataforma de execução, tanto no que diz respeito à portabilidade de código quanto aos mecanismos de comunicação entre objetos.
- A independência em relação aos mecanismos básicos de simulação seqüencial necessários, como escalonador, *threads*, etc., e à linguagem de implementação dos mesmos.
- A independência em relação aos mecanismos de sincronização distribuída empregados, sejam eles pessimistas, otimistas ou híbridos.
- O desempenho do suporte, pois afinal de contas o principal objetivo na distribuição de simulações é acelerar sua velocidade de execução.

Pudemos obter uma boa independência em relação à plataforma de execução pela escolha de uma plataforma aberta nos modelos do padrão CORBA, apresentada no capítulo 4. Essa plataforma vai nos permitir abstrair não só a localização dos objetos remotos na comunicação como também ocultar características internas a cada máquina, como linguagem de implementação dos objetos locais, formato das variáveis, etc.

Para buscar a independência em relação aos mecanismos básicos de simulação seqüencial necessários, faremos uso da biblioteca de simulação seqüencial a objetos C++SIM da

forma mais "*standard*" possível, isto é, usando para as atividades de gerenciamento e sincronização da simulação apenas os serviços básicos de simulação oferecidos pela maioria das bibliotecas, como pedidos de inclusão ou retirada do escalonador, leitura do relógio local, etc. Isto não impede no entanto objetos do modelo de simulação de fazerem uso de recursos mais sofisticados oferecidos pela biblioteca em uso.

Nossa estrutura deve ser relativamente independente dos mecanismos de sincronização distribuída utilizados. Este é um objetivo de difícil alcance, pois cada abordagem de sincronização tem suas peculiaridades em relação à interação com o modelo e com os mecanismos de simulação seqüencial empregados, como vimos no capítulo 2. Assim, um mecanismo de sincronização pessimista precisa ter acesso à informação de previsão (*lookahead*) de cada objeto do modelo. Já um mecanismo de sincronização otimista deve ter condições de intervir diretamente no escalonador local, para executar o retorno em caso de violação de causalidade, e para isso também deve poder salvar e restaurar os estados anteriores dos objetos.

Inicialmente vamos apresentar a estrutura geral da nossa proposta, para em seguida analisar em detalhe o comportamento de cada módulo e as interações entre eles.

5.2 Estrutura Geral do Suporte de Simulação

Consideramos como plataforma física de execução um conjunto de máquinas sem memória comum, comunicando através de uma rede local. Todos os componentes desse sistema são considerados *a priori* confiáveis, e portanto não abordaremos questões relativas a eventuais falhas dos mesmos.

Temos um conjunto de objetos de simulação a executar em um grupo de máquinas. Cada máquina deve então se ocupar da execução de um grupo de objetos, e comunicar com as demais máquinas para as ações de sincronizacão e a ativação de métodos em objetos remotos. Para otimizar a gestão do tempo simulado, os objetos situados em uma mesma máquina ficarão sob o controle de um mesmo escalonador. Desta forma, cada máquina conterá um "subsimulador", responsável pela simulação de seus objetos locais e pelas interações com os demais subsimuladores, para a troca de eventos (ativações de métodos entre objetos locais e distantes) e a sincronização no tempo simulado. Essa estrutura em subsimuladores é inspirada no sistema PROSIT [MM94, FMSM94], apresentado na seção 3.3.2. A figura 5.1 apresenta um esboço dessa estrutura:

Cada subsimulador executa assim uma simulação seqüencial envolvendo seus objetos locais e eventos provindos de outros subsimuladores, que devem ser corretamente gerenciados para evitar ou corrigir violações locais da causalidade.

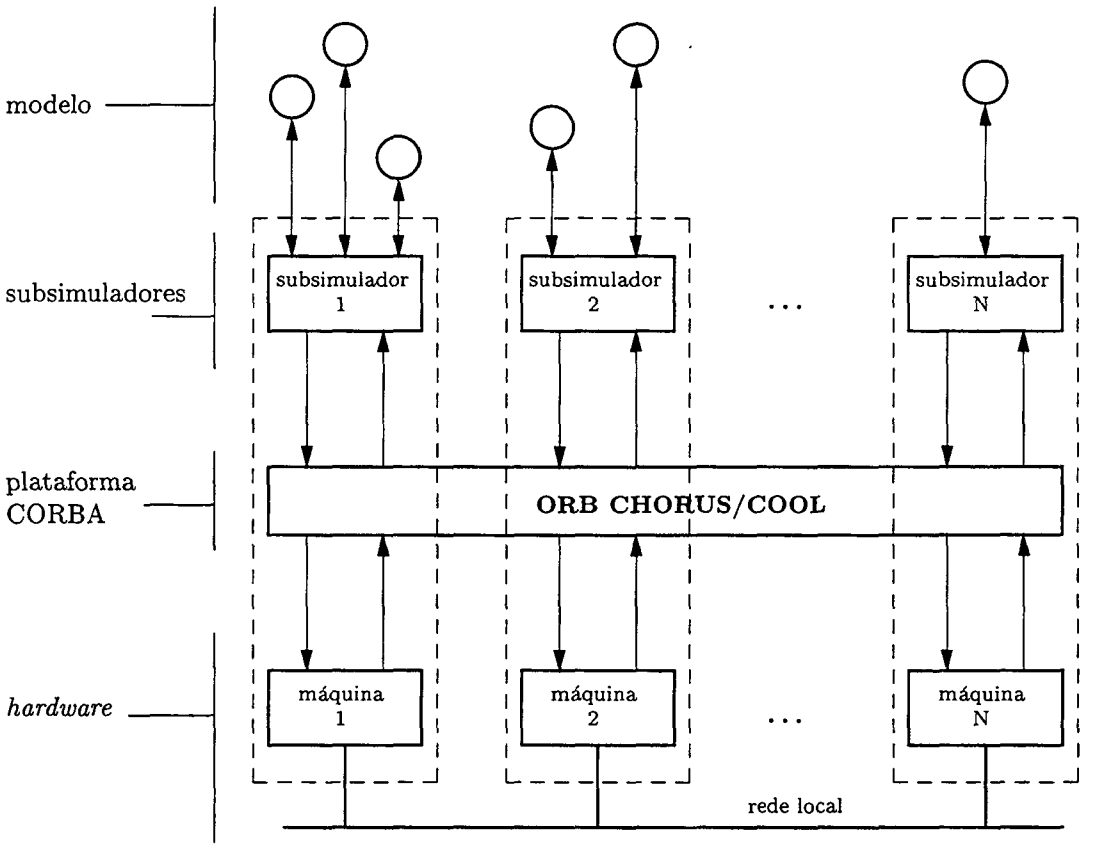

Figura 5.1: Estrutura geral do suporte de simulação

Cada subsimulador é constituído por um único processo UNIX, visando assim minimizar trocas de contexto desnecessárias e proporcionar uma melhor integração com as bibliotecas de simulação seqüencial disponíveis. Esta característica, aliada ao fato de que cada subsimulador executa uma simulação seqüencial coordenada por um relógio local único, permite que os objetos locais façam uso de outras formas de interação além da chamada de métodos, como por exemplo variáveis ou objetos passivos compartilhados, sem prejuízo da causalidade. A interação entre objetos distantes permanece no entanto limitada à ativação de métodos, devido ao assincronismo entre subsimuladores.

Além dos objetos do modelo, cada subsimulador tem sob sua responsabilidade objetos adicionais para o gerenciamento da sincronização distribuída e da troca de eventos entre subsimuladores. Esses objetos devem em princípio ser tratados (escalonados) da mesma forma que os objetos locais do modelo de simulação, para simplificar a estrutura do subsimulador e não sacrificar sua portabilidade.

A biblioteca de simulação C++SIM, apresentada no capítulo 3, serve de base para a construção de cada subsimulador, fornecendo os mecanismos básicos necessários ao geren-

ciamento da simulação seqüencial local. Todavia, o uso de outra biblioteca de simulação seqüencial orientada a objetos, como por exemplo CNCL [JSB+96], não deve apresentar grandes dificuldades, haja visto que os objetos de gerenciamento são tratados da mesma forma que os demais objetos do modelo, e procuram fazer uso somente de serviços comuns, como inserção ou remoção de eventos no escalonador, leitura do relógio local, etc.

Os subsimuladores interagem entre si através da plataforma CORBA. Nosso uso dessa plataforma é relativamente modesto, pois utilizamos somente os serviços de ativação de métodos remotos. Para simplificar os mecanismos de sincronização, consideramos que todas as interações remotas se dão através de chamadas assíncronas (*one-way*). Com isso evitamos a possibilidade de bloqueios ou sincronizações excessivas em chamadas cíclicas, simplificando consideravelmente o gerenciamento da sincronização distribuída.

As interações entre objetos locais a um subsimulador podem ser feitas localmente, sem o auxílio do ORB, para não interferir negativamente no desempenho do sistema. Desta forma, precisaremos definir em cada subsimulador somente as interfaces IDL para os serviços de sincronização distribuída e de troca de eventos remotos.

Nas próximas seções apresentaremos a estrutura interna de cada subsimulador e as interações entre eles.

5.3 Estrutura de um Subsimulador

Como descrito na seção anterior, cada máquina executa um subsimulador responsável pela simulação seqüencial dos objetos locais, pelas trocas de eventos com outros subsimuladores e pelas atividades de sincronização distribuída para a evolução do tempo simulado. Cada subsimulador é composto de:

- *Mecanismos de simulação seqüencial*, basicamente o escalonador e seus serviços associados. Essas funcionalidades são providas pela biblioteca de simulação seqüencial empregada (neste caso C++SIM).

- *Objetos de simulação locais*, que representam a parcela do modelo de simulação alocada àquela máquina.

- *Objeto sincronizador*, responsável pela coleta de informações sobre a simulação local (relógios das entradas e saídas, *lookaheads*, estados, etc.) e pela implementação de uma estratégia de sincronização em conjunto com os objetos sincronizadores dos demais subsimuladores. O sincronizador deve também interagir com o escalonador para controlar a evolução do relógio de simulação local, de acordo com a estratégia de sincronização escolhida.

- *Objeto distribuidor de eventos*, que se ocupa do envio de eventos (ativações de métodos) a objetos distantes e da recepção e execução de pedidos de ativações vindos de outros subsimuladores. O distribuidor de eventos efetua a estampilhagem dos eventos emitidos a outros subsimuladores, notifica o sincronizador sobre emissões e recepções de eventos e escalona adequadamente os eventos externos recebidos para ativação no instante oportuno (ou seja, nas datas de suas estampilhas).

Na figura 5.2 apresentamos a estrutura interna de um subsimulador, com os elementos acima descritos e suas principais interações:

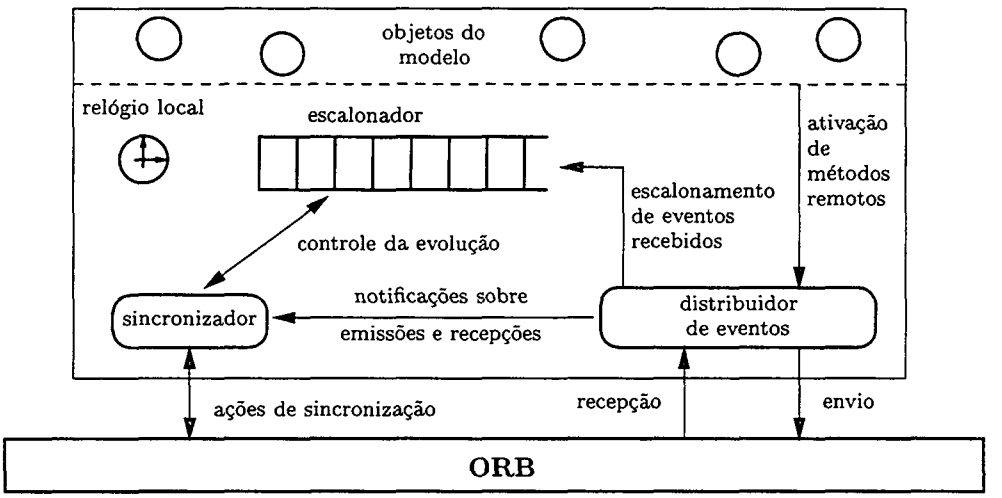

Figura 5.2: Estrutura interna de um subsimulador.

O núcleo do subsimulador é constituído pelo mecanismo escalonador provido pela biblioteca de simulação seqüencial. Todos os objetos locais, inclusive o sincronizador e o distribuidor de eventos, têm sua execução gerenciada pelo escalonador. A interface IDL do subsimulador, que deve ser declarada para seu acesso via ORB, é definida pelos serviços externos oferecidos pelos objetos sincronizador e distribuidor de eventos. Veremos nas próximas seções esses objetos e suas interfaces em detalhe.

5.4 Distribuidor de Eventos

Na estrutura baseada em subsimuladores proposta, temos duas situações distintas para a ativação de métodos entre objetos. A figura 5.3 ilustra essas situações, considerando um objeto `Obj1` ativando o método `met` de um objeto `Obj2`.

Caso os objetos se encontrem no mesmo subsimulador, estarão sob o controle do mesmo escalonador e portanto sincronizados pelo mesmo relógio local. Nesta situação a

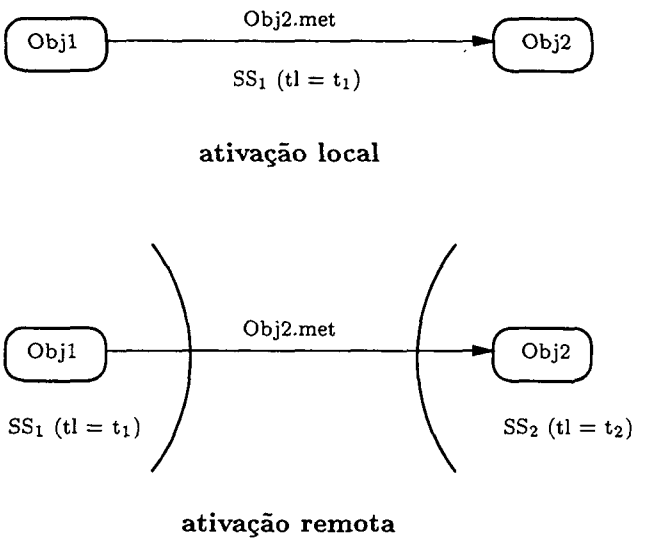

Figura 5.3: Ativação de métodos entre objetos.

ativação de `Obj2.met` pode ser efetuada diretamente, sem interferências ou gerenciamentos específicos.

Na segunda situação os objetos se encontram em subsimuladores distintos, e a ativação de métodos entre eles se torna mais complexa. Para que `Obj1`, situado no subsimulador SS_1 com relógio local t_1, possa ativar um método do objeto `Obj2`, que se encontra no subsimulador SS_2 com relógio local t_2, deveremos:

1. Enviar o pedido de ativação de `Obj2.met` ao subsimulador onde ele se encontra (SS_2), devidamente estampilhado com sua data de ocorrência (t_1).

2. Ao receber o pedido de ativação em SS_2, escaloná-lo para execução em `tl` = t_1.

Os mecanismos de sincronização entre subsimuladores devem garantir que $t_1 \geq t_2$, caso contrário o pedido de ativação estará atrasado em relação à simulação local em SS_2, provocando assim uma violação do princípio de causalidade (que deverá ser detectada e tratada pelo sincronizador de SS_2).

As atividades ligadas à transferência de ativações de métodos entre subsimuladores estão a cargo de um objeto particular do subsimulador, denominado *distribuidor de eventos*. As funções do distribuidor de eventos são:

- Enviar e receber através do ORB, pedidos de ativação de métodos na forma de n-uplas `[data, origem, destino, método, parâmetros]`.

- Estampilhar os pedidos enviados com a data atual de simulação (relógio local do subsimulador de origem).

- Escalonar os pedidos recebidos para execução na data correta (data de ocorrência do evento), no escalonador da simulação seqüencial local.
- Informar o sincronizador sobre os envios e recepções de pedidos, com as respectivas datas, origens e destinos.

A figura 5.4 indica o percurso típico de um pedido de ativação de método entre dois objetos distantes na estrutura proposta. As etapas desse percurso são:

1. O pedido de ativação é emitido pelo objeto `Obj1` ao seu distribuidor de eventos local, na forma `[destino, método, parâmetros]`.
2. O distribuidor de eventos notifica o sincronizador local o envio de um evento, na forma `[data, destino]`.
3. O distribuidor de eventos local envia ao distribuidor de eventos do subsimulador destino o pedido de ativação, na forma `[data, origem, destino, método, parâmetros]`.
4. Ao receber o pedido, o distribuidor de eventos distante notifica ao seu sincronizador a recepção de um evento, na forma `[data, origem]`.
5. O distribuidor de eventos distante escalona o pedido recebido para execução em tl_2 = `data`.
6. O escalonador local avança até tl_2 = `data` e finalmente executa a ativação de método solicitada.

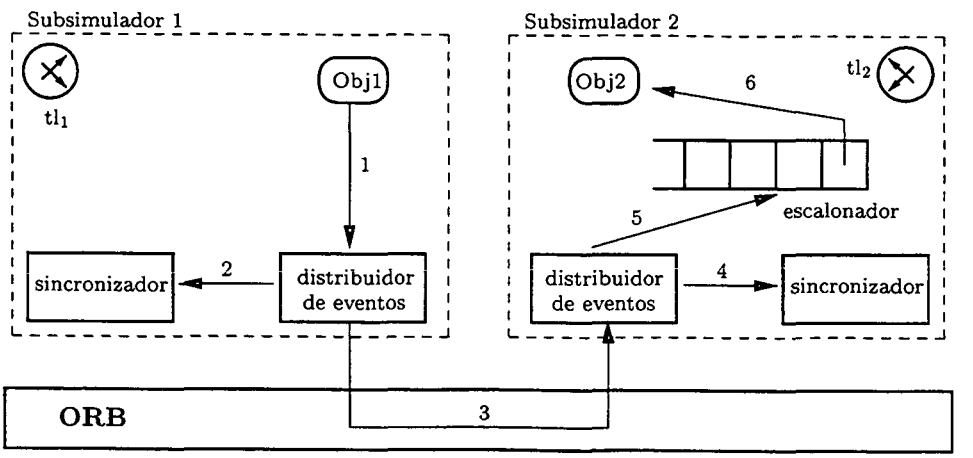

Figura 5.4: Ativação de método remoto.

Na seqüência de operações descritas acima cabem algumas observações relativas a detalhes de implementação. O serviço de recepção do distribuidor de eventos é encarregado do escalonamento de cada evento recebido no escalonador da simulação local. Entretanto, na maioria das bibliotecas de simulação um objeto só pode escalonar a si próprio, não a outros objetos. Essa restrição nos leva à seguinte implementação para o serviço de recepção de eventos:

```
recebe_evento (data,origem,destino,método,parâmetros)
início
   // informa o sincronizador local
   sincronizador.notifica_recepção (data,origem);
   //escalona-se para continuar quando tl = data
   hold (tl = data);
   //executar o pedido de ativação
   executar destino.método (parâmetros);
fim
```

No código acima observamos que o serviço de recepção de eventos, ativado pelo ORB na chegada de um evento ao subsimulador local, suspende-se no escalonador até que o pedido de ativação recebido possa ser executado (`tl` $\geq$ `data`).

Durante esse período, novos pedidos de ativação de métodos podem chegar ao subsimulador. Para evitar prejuízos ao desempenho do sistema, é necessário que o ORB tenha suporte a *threads*, criando uma ativação independente do serviço `recebe_evento` para cada evento entregue ao subsimulador. Assim, o receptor de eventos poderá estar presente simultaneamente em diversas datas no escalonador, uma para cada evento recebido, sem provocar retardos desnecessários na recepção e tratamento de novos eventos externos.

5.5 Sincronização entre subsimuladores

Para que a simulação distribuída possa progredir corretamente, além de executar simulações com seus objetos locais e trocar eventos entre si, os subsimuladores precisam interagir para manter a coerência de seus relógios locais e garantir a correção causal da simulação, através de uma técnica de sincronização como as que vimos no capítulo 2.

O elemento responsável pela manutenção de uma estratégia de sincronizacão em cada subsimulador é o *objeto sincronizador*. Esse objeto deve coletar as informações locais necessárias e interagir com os demais sincronizadores para implementar uma estratégia de sincronização pessimista ou otimista, agindo sobre seu subsimulador de modo a garantir a evolução correta da simulação.

Embora exerçam função equivalente, implementações pessimistas e otimistas de sincronizadores têm estruturas internas bastante distintas e por isso serão abordadas em separado na seqüencia deste texto.

5.5.1 Um Sincronizador Pessimista

Vamos apresentar a estrutura e o comportamento de um sincronizador pessimista baseado na técnica de prevenção de bloqueios via mensagens nulas (capítulo 2). A adaptação desta estrutura a outro esquema de sincronização pessimista não deve apresentar maiores dificuldades. Um sincronizador pessimista tem por funções:

- Gerenciar os relógios dos canais de entrada e de saída do subsimulador. Como vimos no capítulo 2, o mínimo dos relógios dos canais de entrada irá definir o *relógio de entrada* do subsimulador.
- Detectar violações locais da causalidade, que neste caso, são consideradas erros fatais, implicando no encerramento da simulação.
- Coletar previsões (*lookaheads*) do modelo, para atualizar os relógios dos canais de saída.
- Enviar e receber mensagens nulas, com o objetivo de propagar as previsões locais e avançar o relógio de entrada do subsimulador.
- Coordenar a evolução do escalonador local em função do relógio de entrada do subsimulador (o relógio local não deve ultrapassar o relógio de entrada, devido ao risco de ignorar possíveis eventos externos).

A manutenção dos relógios dos canais de entrada e de saída e do relógio de entrada, assim como a detecção de eventuais violações da causalidade, podem ser facilmente implementadas com base nas notificações de envio e recepção de eventos externos, emitidas pelo distribuidor de eventos do subsimulador. Como as notificações de envio têm a forma `[data, destino]` e as de recepção têm a forma `[data, origem]`, o gerenciamento dos relógios dos canais torna-se simples. A validade causal de um envio ou recepção pode ser verificada comparando-se sua data com o relógio do respectivo canal: como os canais são FIFO, o relógio do canal deve ser inferior ou igual à data do evento em questão.

A coleta de previsões dos objetos pode ser efetuada através de um método oferecido pelo sincronizador aos objetos locais, que devem utilizá-lo para atualizar periodicamente suas previsões. A determinação automática de previsões está fora do contexto deste

trabalho, sendo abordada em [Fuj88, Meh91]. O envio de mensagens nulas é feito com base nos relógios dos canais de saída e nas previsões coletadas dos objetos locais, como vimos no capítulo 2. As mensagens nulas recebidas servem unicamente para atualizar os relógios dos canais de entrada, sendo descartadas em seguida.

Para coordenar a evolução do escalonador local em função do relógio de entrada do subsimulador, fazemos uso de uma estratégia simples, mas bastante eficaz. Como o sincronizador é um objeto ativo cuja execução é controlada pelo escalonador, fazemos com que ele seja continuamente escalonado para ativação na data correspondente ao valor atual do relógio de entrada do subsimulador. Assim, o sincronizador irá de certa forma "monopolizar" o escalonador, somente liberando a execução de outros objetos quando suas datas de ativação forem inferiores ou iguais ao relógio de entrada.

Vejamos o comportamento acima descrito em um exemplo. O subsimulador da figura 5.5 tem relógio local `tl = 17` e relógio de entrada `te = 31`. O sincronizador está portanto escalonado para ser ativado em `tl = 31`, e permanecerá assim até que `te` seja incrementado pela chegada de novas mensagens. Nessa situação, somente os eventos escalonados em $t \leq 31$ serão executados antes que o sincronizador retome a posse do escalonador local.

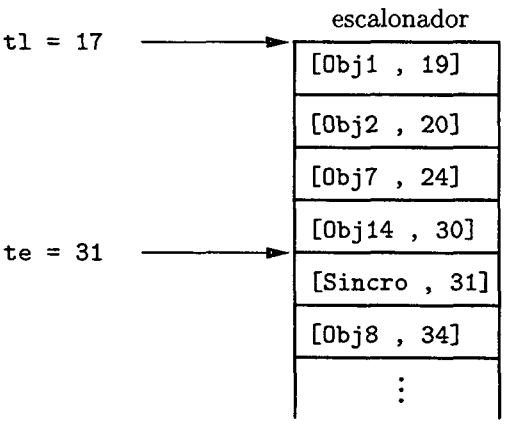

Figura 5.5: Controle do escalonador.

Com base nas descrições acima podemos estabelecer o comportamento básico do objeto sincronizador pessimista e de seus principais serviços em pseudo-código:

```
objeto sincronizador
início    // corpo principal do objeto
   repetir
      envia_msgs_nulas; // sincroniza com outros subsimuladores
      te:= min_i (relógio_entrada [i]); // recalcula relógio de entrada
      hold (tl = te);     // reescalona-se para ativação futura
   até o final da simulação;
fim

método sincronizador.notifica_recepção (data,origem)
início   // informa sincronizador da recepção de evento externo
   se relógio_entrada [origem] > data então
      erro: violação de causalidade;
   senão
      relógio_entrada [origem] := data;
   fim se
fim

método sincronizador.notifica_emissão (data,destino)
início   // informa sincronizador do envio de evento externo
   se relógio_saída [destino] > data então
      erro: violação de causalidade
   senão
      relógio_saída [destino] := data;
   fim
fim

método sincronizador.envia_msgs_nulas
início    // envio de mensagens nulas nos canais de saída
   previsão := relógio_local + lookahead;
   para cada destino D faça
      se relógio_saída [D] < previsão então
         relógio_saída [D] := previsão;
         envia [nula,previsão] ao sincronizador em D;
      fim se
   fim
fim
```

Esta estratégia torna o acoplamento entre sincronizador e escalonador completamente transparente, sem necessidade de alterações ou adaptações neste último. Além disso, o esquema proposto para o acoplamento entre sincronizador e escalonador ajusta-se automaticamente à carga de trabalho imposta a cada subsimulador. Assim, caso muitos eventos sejam escalonados localmente, a ativação do sincronizador será esporádica, para enviar as mensagens nulas necessárias e atualizar o relógio da entrada. Por outro lado, se poucos eventos locais forem escalonados, o sincronizador será constantemente ativado, para que qualquer mudança no relógio de entrada seja rapidamente considerada.

A implementação dos demais serviços do sincronizador pessimista não deve oferecer maiores dificuldades do ponto de vista conceitual. Além disso, a adaptação da estrutura do sincronizador a outra abordagem pessimista é relativamente simples.

5.5.2 Um Sincronizador Otimista

Em princípio a estrutura de simulação apresentada pode aceitar diversas estratégias de sincronização, todavia dificuldades podem ser encontradas no relacionamento entre as estratégias de sincronização otimistas e os mecanismos de escalonamento oferecidos pela biblioteca de simulação seqüencial empregada.

Em caso de detecção de violação do princípio de causalidade, um sincronizador otimista deve agir sobre o escalonador retrocedendo o relógio local a uma data anterior à ocorrência do erro, restaurando a fila do escalonador e os estados dos objetos naquela data e cancelando as mensagens enviadas incorretamente. Vejamos como abordar cada uma destas tarefas:

- *Salvar e restaurar estados de objetos*: para tal podemos fazer uso da técnica de transferência de estados entre réplicas de processos no sistema Isis [BA85]. Nela, cada processo define duas funções para transferência de estados, que podem ser ativadas pelo suporte: *GetState*, que permite obter uma cópia do estado atual do processo, e *SetState*, que permite atribuir um novo estado ao processo. Podemos assim criar métodos similares para cada objeto de simulação e usá-los para armazenar uma seqüência de estados anteriores para cada objeto, restaurando um estado anterior em caso de retorno no tempo simulado.

- *Cancelar as mensagens enviadas*: o sincronizador é notificado pelo distribuidor de eventos a cada envio de evento a outro subsimulador. Desta forma pode ser constituída uma lista de envios para possíveis cancelamentos futuros. As interações entre objetos locais são automaticamente canceladas pelo retorno dos estados dos objetos.

- *Retornar o relógio local e restaurar a fila do escalonador*: estas duas tarefas são bastante delicadas, pois podem implicar em modificações nos mecanismos do escalonador, cuja complexidade depende da estrutura da biblioteca de simulação seqüencial empregada. Neste ponto reside a maior complexidade na implementação de um sincronizador otimista.

- *Calcular o Tempo Virtual Global*: este valor serve para minimizar o número de estados e eventos armazenados em um subsimulador. O TVG é uma propriedade global estável (monotonicamente crescente) e pode ser facilmente calculado por algoritmos clássicos [Ray92].

Assim, podemos concluir que a implementação de um sincronizador otimista é significativamente mais complexa que a de seu equivalente pessimista, e pode limitar a escolha da biblioteca de simulação usada para prover os mecanismos básicos de simulação seqüencial. Por sua vez, o sincronizador pessimista encaixa-se perfeitamente ao escalonador seqüencial, sem necessidade de alterações.

5.6 Conclusão

Neste capítulo apresentamos uma estrutura para o suporte a simulações distribuídas orientadas a objetos sobre uma plataforma CORBA. A estrutura é composta por subsimuladores executando simulações seqüenciais sobre parcelas do modelo e comunicando através dos serviços oferecidos pelo ORB. Cada subsimulador é constituído de uma biblioteca de simulação seqüencial orientada a objetos, objetos ativos do modelo, um objeto distribuidor de eventos (responsável por ativações de métodos entre subsimuladores distintos) e um objeto sincronizador (responsável pela implementação de uma estratégia de sincronização entre subsimuladores).

Vimos que no caso das estratégias de sincronização pessimistas a integração entre os mecanismos de simulação seqüencial e o sincronizador distribuído é totalmente transparente, simples e bastante genérica. Já no caso das abordagens otimistas este ponto apresenta-se delicado, haja visto a necessidade eventual de retroceder o tempo simulado e em conseqüência o escalonador, possibilidade no mínimo pouco usual em simulações seqüenciais. Já os demais serviços necessários às abordagens otimistas podem ser facilmente implementados.

Capítulo 6

Conclusões e Perspectivas

Nesta dissertação foi apresentada uma estrutura que possibilita a execução distribuída de simulações a eventos discretos, construídas sobre um modelo orientado a objetos rodando sobre plataformas heterogêneas. A comunicação entre as plataformas computacionais distintas foi conseguido através do suporte CORBA. Foi analisada a problemática da sincronização decorrente da execução distribuída e verificadas as técnicas propostas na literatura para a solução do problema.

Para conseguir executar simulações tão específicas em sua forma é necessário primeiro definir o suporte a adotar para tal. Para isso atacamos o problema em duas frentes: a simulação a eventos discretos orientada a objetos e a execução distribuída usando CORBA.

Na execução da simulação a eventos discretos orientada a objetos utilizamos uma biblioteca especializada (a biblioteca C++SIM [Uni94]). A mesma se mostrou adequada para as nossas necessidades por apresentar, através de um conjunto de funções simples, as funcionalidades básicas necessárias à construção da estrutura proposta. Com a mesma conseguimos implementar um conjunto de exemplos que nos proporcionaram uma visão ampla sobre suas possibilidades.

Na simulação distribuída sobre CORBA usamos o ORB CHORUS/COOL [Cho96a]. Através do mesmo foi possível a comunicação de objetos distribuídos em sistemas operacionais diferentes como *Solaris* e *Linux*, usados neste trabalho. Também aqui foram construídos pequenos exemplos que nos provaram sua eficácia.

O presente trabalho conclui-se com a proposta de um ambiente para a simulação distribuída orientada a objetos, integrando as duas ferramentas estudadas: C++SIM para os mecanismos básicos de simulação seqüencial em cada máquina e CHORUS/COOL para a integração transparente entre os simuladores. Nossa proposta foi detalhada com a apresentação de sua estrutura interna e dos principais algoritmos envolvidos. Apesar de termos apresentado nossa estrutura baseando-nos na estratégia de sincronização pessimis-

ta de prevenção de bloqueios (mensagens nulas), a estrutura é bastante flexível, podendo ser facilmente adaptada a outras estratégias com reflexos mínimos sobre a mesma. Pode-se mesmo pensar na construção de classes de sincronizadores especializadas para cada estratégia, a exemplo do que ocorre com MOOSE [WB94] e PROSIT [MM94].

Outra característica importante da estrutura proposta é a relativa independência do suporte à simulação seqüencial empregado em cada máquina, no caso a biblioteca C++SIM, ao menos no que diz respeito às estratégias de sincronização pessimistas. De fato, o esquema de interação entre o sincronizador e os demais objetos locais, inclusive o escalonador de eventos, é bastante simples: o sincronizador apresenta-se ao escalonador como apenas mais um objeto a ser escalonado, como os demais. Isto implica na possibilidade de uso de diferentes bibliotecas de simulação, em máquinas distintas. Aliada à presença do ORB, essa propriedade permitiria inclusive a integração entre diferentes linguagens de simulação seqüencial a objetos.

Como perspectivas para a continuidade deste trabalho podemos apontar:

- A implementação, testes e avaliação de desempenho da estrutura proposta.
- O uso desse ambiente na simulação intensiva de problemas reais da área de sistemas a eventos discretos.
- Propiciar mecanismos para implementar a transparência na localização dos objetos de simulação. Atualmente a ativação de um método remoto deve ser dirigida ao distribuidor de eventos, que a repassa ao distribuidor remoto, que por sua vez a escalona para a execução no momento (simulado) oportuno. Acreditamos que, fazendo uso das facilidades do ORB e de técnicas de reflexão computacional [Mae87], seja possível implementar as ativações de métodos remotos de forma completamente transparente ao programador do modelo de simulação, ocultando ou evitando a passagem pelos distribuidores de eventos.
- Por razões de simplicidade, nosso modelo suporta apenas ativações assíncronas de métodos remotos. Seria interessante suportar também ativações síncronas, para aumentar o poder de expressão dos modelos de simulação aceitos.
- Prover classes de sincronizadores implementando diferentes estratégias de sincronização pessimistas, otimistas ou híbridas.
- Efetuar avaliações de desempenho no sentido de verificar se o suporte CORBA é adequado para a forte necessidade de comunicação encontrada normalmente em simulações distribuídas.

Bibliografia

[BA85] K. P. Birman and Joseph T. A. Replication and fault-tolerance in the Isis system. In *ACM SIGOPS*, volume 10, pages 79–86, 1985.

[Bag91] Rajive L. Bagrodia. Designing efficient simulations using Maisie. In *Proceedings of the 1991 Winter Simulation Conference*, pages 243–247, december 1991. Computer Science Departament.

[BDMN73] G. M. Birtwistle, O. J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin.* Chartwell-Bralt Ltd., 1973.

[BT87] G. Bracha and S. Toueg. Distributed deadlock detection. *Distributed Computing*, 1(2):127–138, 1987.

[Cho96a] Chorus Systèmes. *CHORUS/COOL-ORB r3 - Product Description*, June 1996. CS/TR-95-157.3.

[Cho96b] Chorus Systèmes. *CHORUS/COOL-ORB Tutorial*, June 1996. CS/TR-96-4.1.

[CJS87] I. Cidon, J. Jaffe, and M. Sidi. Local distributed deadlock detection by cycle detection and clustering. *IEEE Transactions on Software Engineering*, SE-13(1):3–14, 1987.

[CM79] K. M. Chandy and J. Misra. Distributed simulation: a case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, SE-5(5):440–452, September 1979.

[CM81] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(11):198–206, April 1981.

[FMSM94] Patrick Ferrante, Philippe Mussi, Günther Siegel, and Lionel Mallet. Object oriented simulation: Highlights on the PROSIT parallel discrete event simulator. Rapport de recherche 2235, INRIA, Avril 1994. Projet Mistral.

[Fuj88] R. M. Fujimoto. Lookahead in parallel discrete event simulation. In *Proceedings of the 1988 International Conference on Parallel Processing, Pennsylvania*, pages 34–41, August 1988.

[Fuj90] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):31–53, October 1990.

[Jef85] D. Jefferson. Virtual time. *ACM Toplas*, 7(3):404–425, July 1985.

[JSB+96] M. Junius, M. Steppler, M. Büter, D. Pesch, et al. *CNCL-Communication Networks Class Library.* Aachen University of Technology, Germany, 1.8 edition, 1996.

[LB91] Greg Lomow and Dirk Baezner. A tutorial introduction to object-oriented simulation and Sim++. In *Proceedings of the 1991 Winter Simulation Conference*, pages 157–163, 1991.

[Mae87] Pattie Maes. Concepts and experiments in computacional reflexion. In *OOPSLA'87 Proceedings*, pages 147–156, October 1987.

[Maz94] C. A. Maziero. *Conception et réalisation d'un noyau de système réparti pour la simulation parallèle.* PhD thesis, Université de Rennes 1 - France, 1994.

[MB95] Jay M. Martin and Rajive L. Bagrodia. COMPOSE: An object-oriented environment for parallel discrete-event simulations. In *Proceedings of the 1995 Winter Simulation Conference*, December 1995.

[Meh91] H. Mehl. Speed-up of conservative distributed discret event simulation methods by speculative computing. In *IEEE/ACM/SCS Workshop on parallel and distributed simulation*, Anaheim - California, January 1991.

[Mis86] J. Misra. Distributed discrete-event simulation. *Computing Surveys*, 18(1):39–65, March 1986.

[MM94] Lionel Mallet and Philippe Mussi. Object oriented parallel discrete event simulation: The PROSIT approach. Rapport de recherche 2232, INRIA, Avril 1994. Projet Mistral.

[Obj95] Object Management Group - OMG. *The Common Object Request Broker: Architecture and Specification*, July 1995. Revision 2.0.

[P+94] G. D. Parrington et al. The design and implementation of Arjuna. Technical report, Broadcast Project, Oslo, Norway, October 1994.

[PHB93] H. T. Papadopoulos, C. Heavey, and J. Browne. *Queueing Theory in Manufacturing Systems Analysis.* Chapman & Hall, 1993.

[Ray92] M. Raynal. *Syncronisation et état global dans les systèmes répartis.* Eyrolles, Janvier 1992. Collection EDF.

[RW89] R. Righter and J. C. Walrand. Distributed simulation of discrete event systems. *Proceedings of the IEEE*, 77(1):99–113, January 1989.

[Siq95] Frank Siqueira. *CORBA - Common Object Request Broker Architecture.* Universidade Federal de Santa Catarina, 1995. Laboratório de Controle e Microinformática - LCMI.

[SMP88] B. Samadi, R. R. Muntz, and D. S. Parker. A distributed algorithm to detect a global state of a distributed simulation system. In *IFG WG 10.3, on Distributed Processing*, pages 19–34, North-Holand, 1988. Elsevier Science Publishers B.V.

[SSS88] C.L. Seitz, J. Seizovic, and Wen-King Su. The C programmer's abbreviated guide to multicomputer programming. Technical report, Dept. of Computer Sciences, California Institute of Technology, Los Angeles, January 1988. Caltech-CS-TR-88-1.

[Sun90] V. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4), December 1990.

[Uni94] University of Newcastle Upon Tyne. *C++SIM User's Guide*, 1994. Draft Version 1.0.

[WB94] Jerry Waldorf and Rajive L. Bagrodia. A concurrent object oriented language for simulation. In *International Journal of Computer Simulation*, volume 4(2), pages 235–257, 1994.