Um suporte para simulações orientadas a objetos distribuídas em uma plataforma aberta*

Richard D. Ribeiro, M.Sc.
Depto. de Informática
CEFET-PR
80.230-901 – Curitiba – PR
richard@dainf.cefetpr.br

Carlos A. Maziero, Dr.†
Programa de Pós-Graduação em Informática Aplicada
Pontifícia Universidade Católica do Paraná
80.215-901 – Curitiba – PR
maziero@ccet.pucpr.br

Resumo

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 orientadas 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

This work aims to define the structure of a distributed discrete-event simutation runtime support. To simplify the construction of simulation models, we choosed to use object-oriented languages, thus allowing hierarchical construction of the models, and code reuse. Instead of creating a model description language specific to this simulation support, we used well-known object-oriented simulation libraries, giving us a more generic and portable environment. The problems arising from the use of object-oriented programming model over an heterogeneous distributed computing environment are solved using an open platform for object communication, following the CORBA standards.

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 [4, 16, 11]. 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 dias de processamento em um sistema seqüencial [13].

*Trabalho desenvolvido no Depto. de Automação e Sistemas da Univ. Federal de Santa Catarina – DAS/UFSC.
†Professor adjunto do DAS/UFSC até jan/98.

Nestes casos a paralelização da simulação pode ser de grande utilidade, haja visto a significativa quantidade de paralelismo potencial geralmente presente nos modelos. 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 [13, 16, 7, 11].

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 [2].

Este trabalho tem como objetivo propor uma estrutura para suportar a execução distribuída de simulações orientadas a objetos sobre plataformas computacionais 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 [14, 17].

Este artigo está dividido em sete seções: a seção 2 apresenta os principais conceitos e técnicas empregadas em simulação seqüencial; a seção 3 mostra a simulação ocorrendo em um ambiente distribuído; a seção 4 introduz o uso da orientação a objetos na simulação; a seção 5 apresenta a arquitetura CORBA; a seção 6 apresenta a estrutura proposta para a simulação distribuída de modelos construídos segundo o paradigma de orientação a objetos e finalmente a seção 7 apresenta as conclusões finais deste trabalho.

2 Simulação a Eventos Discretos

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", "tempo lógico", ou "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. Tal evolução deve respeitar os dois princípios básicos dos sistemas físicos: a *causalidade* e o *determinismo*, segundo o quais o futuro não pode influenciar o passado, e o comportamento futuro pode ser determinado a partir do passado e do presente do sistema.

Na simulação a eventos discretos devemos controlar a ordem dos eventos a serem tratados 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 um escalonador, ou seja, uma fila na qual os eventos a processar são ordenados de forma crescente segundo a data de ocorrência de cada um deles. 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 a ordenação crescente). Os demais eventos do escalonador são considerados potenciais, pois podem ser modificados ou cancelados no tratamento dos eventos anteriores.

3 Simulação Distribuída

A distribuição da simulação sobre um conjunto de processadores pode aumentar em muito a velocidade de execução do simulador nos casos de modelos de grandes dimensões, 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 [3, 4, 13, 16, 7, 11].

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 1).

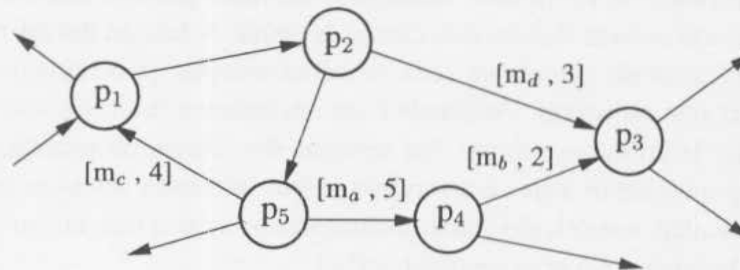

Figura 1: Modelo de simulação.

A evolução do tempo simulado deve seguir uma referência única, como ocorre no sistema real, mas o contexto paralelo permite que cada processo se comporte como um simulador seqüencial quase independente. Desta forma podemos adotar uma estratégia assíncrona para a evolução do tempo simulado, onde cada processo gerencia uma cópia local do relógio de simulação, chamada *relógio local*, que evolui em função do estado local do processo [16]. 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 artigo [4] 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. Então cada processo deve tratar as mensagens recebidas em seus canais de entrada na ordem das estampilhas. Entretanto, quando houverem canais vazios (sem mensagem recebida) na entrada do processo, este terá dificuldade para estabelecer a ordem correta das mensagens a consumir, pois novas mensagens podem chegar nesses canais vazios.

Para resolver esse problema duas classes de estratégia foram propostas [7]:

- *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.

- *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.

3.1 Prevenção de Bloqueios

Uma proposta do tipo pessimista para prevenir bloqueios utiliza mensagens de controle para evitar que tais situações ocorram. Essas mensagens são chamadas de "mensagens nulas" [3, 11].

Ao enviá-las, um processo comunica a outro uma "previsão" (*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 (no tempo simulado), o comportamento do processo, etc.

Toda vez que um processo envia uma mensagem não-nula por um canal de saída, ele envia também mensagens nulas através dos demais canais de saída. A função dessas mensagens é fazer com que o relógio do canal de entrada de cada processo receptor (e o relógio do canal de saída do processor emissor) seja avançado. 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 [11].

4 Simulação Orientada a 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, um 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.

4.1 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 heterogêneo implica em:

- Mapear modelos expressos em termos de objetos e métodos em um modelo computacional mais adequado 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 [19], Sim++ [8] e outras baseadas em bibliotecas de uma linguagem convencional, como COMPOSE [10] e PROSIT [9].

Os suportes distribuídos para a simulação orientada a objetos atualmente existentes incorporam seus próprios mecanismos de resolução (limitada) da heterogeneidade, quando não a ignoram simplesmente. Nosso trabalho visa a definição de um ambiente de simulação a eventos discretos orientada a objetos sobre uma plataforma aberta. Desta forma, optamos por empregar uma plataforma de execução nos moldes da norma CORBA, e sobre ela construir um ambiente de simulação orientado a objetos.

5 A Arquitetura CORBA

CORBA é uma especificação de arquitetura proposta pela OMG [14] para permitir a interoperabilidade de *softwares* orientados a objetos em ambientes distribuídos heterogêneos. A arquitetura CORBA é composta por um núcleo chamado ORB (*Object Request Broker*, responsável pela comunicação entre os vários objetos distribuídos), pelas implementações de objetos criados pelo usuário, pelas interfaces estáticas e dinâmicas de invocação daqueles objetos, pelos adaptadores de objetos e pelos repositórios de interface e de implementações. A figura 2 apresenta de maneira simplificada a arquitetura CORBA.

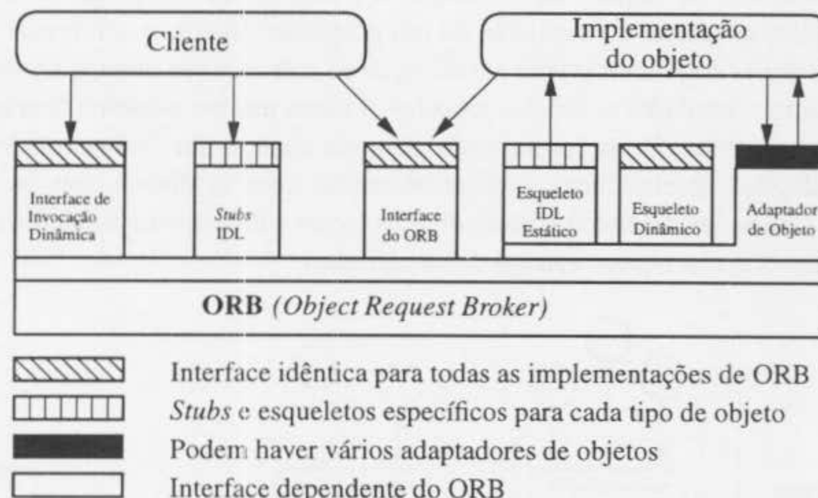

Figura 2: A arquitetura CORBA

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 acesso dinâmico (em tempo de execução).

A invocação de métodos em objetos remotos pode ser efetuada de duas formas. Um objeto pode fazer uso das *stubs* geradas na compilação da descrição de interface (arquivo IDL) do objeto remoto, no caso de ter acesso às mesmas. Pode-se também utilizar a interface de invocação dinâmica (*Dynamic Invocation Interface* - DII) para construir e expedir dinamicamente invocações remotas. Para que isto seja possível, uma descrição da interface do objeto deve ter sido adicionada

previamente ao repositório de interfaces, permitindo acesso aos métodos do objeto através desse serviço durante a execução. Este último mecanismo apresenta a vantagem de permitir a construção de invocações de métodos em objetos desconhecidos durante a compilação do objeto cliente, mas implica em um custo adicional de processamento para a construção das invocações dinâmicas [14].

Ao receber 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, os valores de saída e possíveis exceções são retornados ao cliente.

6 O Suporte a Simulação Proposto

6.1 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. A figura 3 apresenta um esboço dessa estrutura:

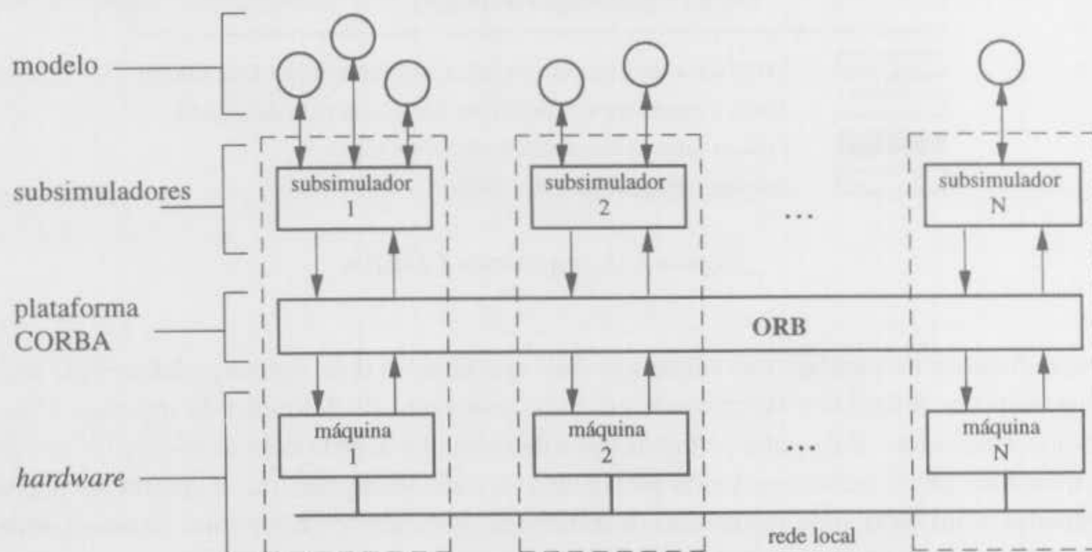

Figura 3: Estrutura geral do suporte de simulação

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.

Cada subsimulador é constituído por um único processo (ou seja, um espaço de endereços comum), 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 seqüencial serve de base para a construção de cada subsimulador, fornecendo os mecanismos básicos necessários ao gerenciamento da simulação seqüencial local. 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. Isso permite que uma biblioteca de simulação seqüencial orientada a objetos possa ser substiuída por outra sem grandes dificuldades. Na construção de nosso protótipo estamos usando a biblioteca de simulação seqüencial orientada a objetos *C++SIM* [18].

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, precisamos 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.

6.2 Estrutura de um Subsimulador

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.
- *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 4 apresentamos a estrutura interna de um subsimulador, com os elementos acima descritos e suas principais interações:

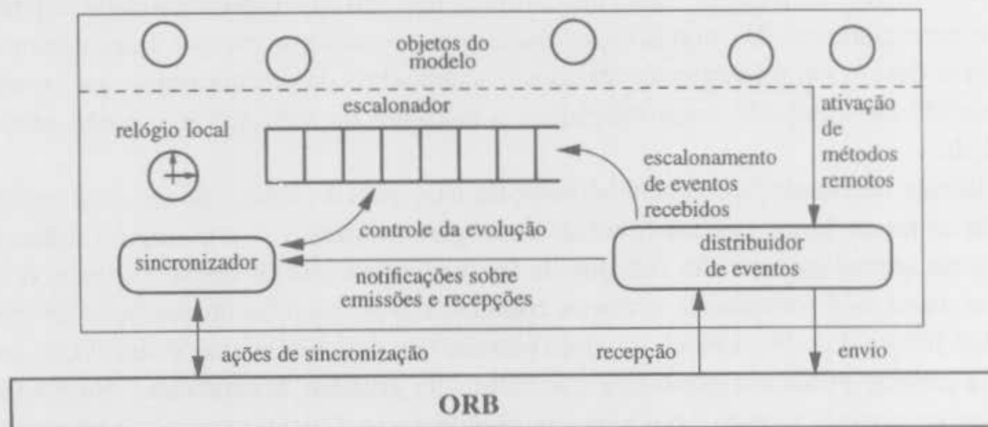

Figura 4: Estrutura interna de um subsimulador.

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 declaıada 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.

6.3 Distribuidor de Eventos

Na estrutura baseada em subsimuladores proposta, temos duas situações distintas para a ativação de métodos entre objetos: 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 ativação de métodos entre objetos 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 deve ser feita através do envio do pedido de ativação ao subsimulador onde se encontra o objeto distante, devidamente estampilhado com a sua data de ocorrência; ao receber tal pedido o subsimulador deve escaloná-lo para a execução na data correta, garantindo que essa execução não viole a causalidade.

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.

Na seqüência de operações descritas acima cabem algumas observações relativas à 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:

```
método 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`). Desta forma, aquela instância do método `recebe_evento` torna-se responsável pela execução do evento recebido no momento apropriado.

Durante o período em que o método `recebe_evento` está suspenso, novos pedidos de ativação de métodos podem chegar ao subsimulador, contendo datas de ativação diversas (inclusive anteriores à do último pedido recebido, vindas de outros subsimuladores). Esses novos pedidos devem ser incluidos no escalonador local assim que recebidos, para que os respectivos eventos sejam considerados na simulação local.

Caso o receptor de eventos permita processar apenas um evento externo por vez, os novos eventos recebidos devem esperar até que a execução do último evento seja completada. Essa condição pode levar a uma considerável degradação no desempenho do sistema, e até mesmo a situações de bloqueio, caso seja usado um sincronizador pessimista[1].

Desta forma, é essencial que a plataforma ORB utilizada ofereça suporte a *threads*, permitindo criar uma ativação independente do serviço `recebe_evento` para cada novo evento entregue ao subsimulador. Com isso, o receptor de eventos poderá estar presente simultaneamente em diversas datas no escalonador, uma para cada evento recebido. Assim, novos eventos serão inclusos no escalonador local assim que recebidos, evitando retardos desnecessários e possíveis situações de bloqueio.

6.4 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 apresentadas na seção 3.

[1]Caso os novos eventos recebidos sejam necessários para o avanço do relógio do subsimulador local, é possível que a data de ocorrência do último evento nunca seja alcançada, bloqueando a simulação.

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.

6.4.1 Um Sincronizador Pessimista

Vamos apresentar a estrutura e o comportamento de um sincronizador pessimista baseado na técnica de prevenção de bloqueios via mensagens nulas [3, 11]. Um sincronizador pessimista tem por funções:

- gerenciar os relógios dos canais de entrada e de saída 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 [6, 12]. O envio de mensagens nulas é feito com base nos relógios dos canais de saída e nas previsões coletadas dos objetos locais. 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 locais quando suas datas de ativação forem inferiores ou iguais ao relógio de entrada do subsimulador, garantindo assim o respeito à causalidade entre eventos locais e externos.

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 estejam escalonados localmente (muita atividade local), 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 estiverem escalonados (pouca atividade local), o sincronizador será constantemente ativado, para que qualquer mudança no relógio de entrada seja rapidamente considerada.

6.4.2 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 de técnicas de transferência de estados entre réplicas de processos como aquelas usadas no ambiente Isis [1]. Neste último, cada processo define duas funções para transferência de estados, que podem ser ativadas pelo suporte de execução: *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. Com esses dados coletados pode ser construída uma lista de envios para possíveis cancelamentos futuros. As interações entre objetos locais são automaticamente canceladas restaurando os estados anteriores 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*: O TVG permite limitar o número de estados e eventos a ser armazenados por um subsimulador para possibilitar retornos a estados anteriores. Seu valor pode ser visto como uma propriedade global estável (monotonicamente crescente), e pode ser facilmente calculado por algoritmos clássicos [15].

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.

7 Conclusão

Neste trabalho foi apresentada a arquitetura de um suporte para a execução distribuída de simulações a eventos discretos, construídas sobre um modelo orientado a objetos rodando sobre plataformas heterogêneas. Essa estrutura procura dar resposta a diversos aspectos envolvidos na execução de simulações em um contexto distribuído: o uso da orientação a objetos em simulações a eventos discretos, a execução de aplicações orientadas a objetos em um contexto distribuído e também os mecanismos de sincronização necessários à execução distribuída de simulações.

Nossa proposta de ambiente integra duas ferramentas: uma biblioteca especializada para os mecanismos básicos de simulação seqüencial em cada máquina e o suporte CORBA para a integração transparente entre os subsimuladores. Na execução da simulação a eventos discretos orientada a objetos utilizamos a biblioteca C++SIM [18]. Esta se mostrou adequada para as nossas necessidades por apresentar, através de um conjunto de funções hierárquicas simples, as funcionalidades básicas necessárias à construção da estrutura proposta. A comunicação transparente entre subsimuladores em plataformas computacionais distintas foi conseguida através do suporte CORBA provido pela plataforma *Chorus/COOL* [5].

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 pessimista de prevenção de bloqueios (mensagens nulas), a estrutura é bastante flexível, podendo ser adaptada a outras estratégias com algumas modificações sobre a mesma. Pode-se mesmo pensar na construção de classes de sincronizadores especializadas para cada estratégia, a exemplo do que ocorre com MOOSE [19] e PROSIT [9].

Outra característica importante da estrutura proposta é a relativa independência do suporte à simulação seqüencial empregado em cada máquina, 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 suportes de simulação seqüencial orientada a objetos.

Referências

[1] Birman, K. P., and A., J. T. Replication and fault-tolerance in the Isis system. In *ACM SIGOPS* (1985), vol. 10, pp. 79–86.

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

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

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

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

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

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

[8] Lomow, G., and Baezner, D. A tutorial introduction to object-oriented simulation and Sim++. In *Proceedings of the 1991 Winter Simulation Conference* (1991), pp. 157–163.

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

[10] Martin, J. M., and Bagrodia, R. L. COMPOSE: An object-oriented environment for parallel discrete-event simulations. In *Proceedings of the 1995 Winter Simulation Conference* (December 1995).

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

[12] Mehl, H. 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).

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

[14] OBJECT MANAGEMENT GROUP - OMG. *The Common Object Request Broker: Architecture and Specification*, July 1995. Revision 2.0.

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

[16] RIGHTER, R., AND WALRAND, J. C. Distributed simulation of discrete event systems. *Proceedings of the IEEE 77*, 1 (January 1989), 99–113.

[17] SIQUEIRA, F. *CORBA - Common Object Request Broker Architecture*. Universidade Federal de Santa Catarina, 1995. Laboratório de Controle e Microinformática - LCMI.

[18] UNIVERSITY OF NEWCASTLE UPON TYNE. *C++SIM User's Guide*, 1994. Draft Version 1.0.

[19] WALDORF, J., AND BAGRODIA, R. L. A concurrent object oriented language for simulation. In *International Journal of Computer Simulation* (1994), vol. 4(2), pp. 235–257.