UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ANÁLISE DA DISTRIBUIÇÃO DE UM SIMULADOR MULTINÍVEL

por

Joice Pavek Figueiró

Dissertação submetida como requisito parcial
para obtenção do grau de
Mestre em Ciência da Computação

Prof. Flávio Rech Wagner
Orientador

Porto Alegre, janeiro de 1994.

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Figueiró, Joice Pavek

ANÁLISE DA DISTRIBUIÇÃO DE UM SIMULADOR MULTINÍVEL / Joice Pavek Figueiró. - Porto Alegre: CPGCC da UFRGS, 1994. 78 p.: il.

Dissertação (mestrado)-Universidade Federal do Rio Grande do Sul, Curso ed Pós-Graduação em Ciência da Computação, Porto Alegre, 1994. Orientador: Wagner, Flávio Rech.

Dissertação: CAD para Sistemas Digitais, Simulação de Hardware, Simulação Distribuída, Simulação Multinível.

AGRADECIMENTOS

Após tanto tempo de trabalho, há muito o que agradecer a várias pessoas. No entanto, é impossível aqui mencionar todos que colaboraram, de uma formaou de outra, para que este trabalho se realizasse. Mesmo assim, algumas pessoas merecem menção especial.

Agradeço primeiramente aos professores do CPGCC e do Instituto de Informática, que me acompanham desde a graduação e que sempre me incentivaram a buscar mais do que me era oferecido pelas aulas. Entre todos os professores, um agradecimento especial à Professora Carla Maria Dal Sasso Freitas e ao Professor Lisboa.

Aos colegas do CPGCC, que agora vem se tornando também colegas de trabalho, um grande abraço e obrigada pela amizade e companheirismo. Em particular, guardarei uma lembrança carinhosa dos colegas Soraia, Doris, Edelweiss, Iara, Marinho, Leo, Paulo Wagner e Rafael Bordini.

Ninguém colaborou mais para que esse trabalho se concretizasse do que a família. Sem ela, nem o início teria sido possível, quanto mais o fim. Beijos para Nedir (a mãe), Fernanda e Marcelo.

Finalmente, um agradecimento muito especial ao meu orientador, Prof. Flávio Rech Wagner, cuja correta orientação e total compreensão dos problemas apresentados no decorrer do tempo foram imprescindíveis. Obrigada, Flávio.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

A validação de projetos de sistemas eletrônicos pode ser feita de diversas maneiras, como tem sido mostrado pelas pesquisas em síntese automática e verificação formal. Porém, a simulação ainda é o método mais utilizado.

O projeto de um sistema digital típico pode ser desenvolvido em diversos níveis de abstração, como os níveis algorítmico, lógico ou analógico. Assim, a simulação também deve ser executada em todos esses níveis.

A simulação apresenta, contudo, o inconveniente de não conseguir conciliar uma alta acuracidade de resultados com um baixo tempo de simulação. Quanto mais detalhada é a descrição do circuito, maior é o tempo necessário para simulá-lo. O inverso também é verdadeiro, ou seja, quanto menor for a acuracidade exigida, menor será o tempo dispendido.

A simulação multinível tenta conciliar eficiêencia e acuracidade na simulação de circuitos digitais, propondo que partes do circuito sejam descritas em diferentes níveis de abstração. Com isso, somente as partes mais críticas do sistema são descritas em detalhes e a velocidade da simulação aumenta.

Contudo, essa abordagem não é suficiente para oferecer um grande aumento na velocidade de simulação de grandes circuitos.

Assim, surge como alternativa a aplicação de técnicas de processamento distribuído à simulação multinível.

Os aspectos que envolvem a combinação dessas duas técnicas são abordados nesse trabalho.

Como plataforma para os estudos realizados, optou-se por usar duas ferramentas desenvolvidas nessa Universidade: os simuladores do Sistema AMPLO e o Sistema Operacional HetNOS.

São estudadas técnicas de sincronização em sistemas distribuídos, fundamentais para o desenvolvimento dos simuladores e, finalmente, são propostas alternativas para a distribuição dos simuladores. É realizada, ainda, uma análise comparativa entre as versões propostas.

ABSTRACT

The validation of electronic systems design has been taking different ways, as shown by the investigations on automatic synthesis and formal verification. However, simulation still is the most used method.

The design of a typical digital system can be developed at abstraction levels such as the algorithmic, the logical and the analogical ones. Thus, simulation must run at all these levels.

One inconvenient simulation brings is the difficulty in conciliating accuracy with low time cost. The more detailed a circuit description is, the more time consuming the simulation is and vice-versa.

Multilevel simulation tries to join efficiency with accuracy in digital circuits simulation as long as it proposes different levels of abstraction for the description of the different parts of the circuit. This way, simulation speed increases as detailed descriptions are only devoted to critical parts of the system.

However, this approach is not good enough to offer a great speed-up in the simulation of very complex circuits.

So, multilevel simulation can be helped by distributed processing techniques. This association is studied in this work.

Two software tools were used: the simulators of the AMPLO system and the HetNOS operating system, both developed at Universidade Federal do Rio Grande do Sul.

Syncronization techniques in distributed systems were studied, since they are necessary for the development of distributed simulators. In addition, a concise analysis of the centralized version of the simulators is presented. Finally, different alternatives for the distribution of the simulators are proposed and compared.

1 INTRODUÇÃO

Simulação é ainda o método de validação de projeto de sistemas eletrônicos mais utilizado na prática, apesar dos avanços recentes nas áreas de síntese automática e verificação formal.

Seguindo os passos de um processo típico de projeto, a simulação de um sistema eletrônico digital é realizada em diferentes níveis de abstração, sejam eles discretos, tais como os níveis algoritmico, RT e lógico, sejam analógicos, como o nível de circuitos elétricos. A realização de um projeto numa seqüência de níveis de abstração tenta resolver satisfatoriamente um compromisso entre a eficiência e a acuracidade. A simulação em níveis mais altos de abstração pode ser realizada em tempos de processamento consideravelmente menores do que aqueles exigidos nos níveis inferiores. No entanto, a acuracidade necessária em muitas situações só é obtida nos níveis inferiores de projeto.

Um método que procura conciliar eficiência e acuracidade é a simulação multinível, na qual apenas certas partes do sistema são simuladas em níveis mais detalhados de abstração.

Na UFRGS, foi desenvolvido, no âmbito do projeto AMPLO (AMbiente integrado para Projeto Lógico de sistemas digitais) [FRE88], um ambiente de simulação multinível baseado no princípio mestre-escravo [WAG89]. O ambiente permite a simulação de sistemas digitais

descritos modularmente, nos quais módulos podem estar modelados em três diferentes níveis discretos de abstração: nível de sistema, de transferência entre registradores e nível de portas lógicas.

Cada módulo é simulado através de um simulador "escravo", dedicado ao nível de abstração no qual o módulo está descrito. Um simulador "mestre" coordena as interações entre os módulos e o avanço do tempo de simulação. O ambiente oferece recursos de gestão do processo de simulação e de interação com o usuário que são comuns a todos os níveis de abstração [WAG90].

A modelagem de partes de um sistema em níveis mais altos de abstração não é, no entanto, suficiente para uma redução significativa do tempo de simulação quando se trata de um projeto de sistemas de grande porte.

Acompanhando a evolução das plataformas computacionais, verifica-se que está ocorrendo nos últimos anos um forte desenvolvimento, em todas as áreas de aplicação, de simuladores paralelos, usando técnicas de sistemas distribuídos [MIS86]. Esta tendência se manifesta também na área de sistemas eletrônicos [SMI86].

Um sistema distribuído consiste em uma coleção de processos distintos, separados espacialmente, comunicando-se entre si exclusivamente por meio de troca de mensagens. Isto significa que eles não compartilham variáveis de memória. Cada variável só tem validade

localmente dentro de cada processo. Os tempos de transmissão das mensagens são insignificantes diante dos tempos de processamento dos eventos dos processos.

O controle entre os processos deve também ser distribuído, isto é, não devem existir processos centrais que façam o roteamento das mensagens, nem o direcionamento de operações para outros processos.

A distribuição de uma simulação pode ser feita pelo particionamento do algoritmo de simulação entre diversos processadores, ou pelo particionamento do sistema em subsistemas que são simulados por diferentes processadores. Esta segunda categoria é particularmente apropriada para o desenvolvimento de simuladores distribuídos numa rede de processadores.

Dentro dessa visão, o presente trabalho analisa vários aspectos referentes à criação da versão distribuída dos simuladores do AMPLO. Entre esses aspectos está a proposta de alternativas para distribuir o sistema já existente, considerando as formas de sincronizações possíveis e as conseqüências de cada possibilidade.

Esta análise é detalhada até o nível de troca de mensagens e algoritmos necessários para a implementação das versões, escolhendo, inclusive, uma plataforma de desenvolvimento.

Além disso, o trabalho analisa qualitativamente o desempenho das várias alternativas de implementação propostas, levando em conta as características dos modelos a serem simulados.

No capítulo 2, é detalhado um aspecto fundamental da simulação distribuída: a sincronização. São analisadas as técnicas otimista e conservativa de sincronização.

No capítulo 3, são descritos os principais aspectos da versão centralizada dos simuladores do AMPLO. A descrição não é detalhada, mas são ressaltados os aspectos mais importantes a serem considerados em uma versão distribuída.

No Capítulo 4, são analisadas plataformas de *software* para a implementação das versões distribuídas. São estudados alguns recursos de *software* existentes e escolhido um para a implementação do sistema.

No Capítulo 5, são analisadas as possibilidades de distribuição para os simuladores. As versões distribuídas são classificadas segundo o grau de distribuição dos seus módulos e segundo o método de sincronização utilizado.

No Capítulo 6, é feito o estudo qualitativo do desempenho das versões distribuídas propostas, considerando as características dos modelos a serem simulados.

Finalmente, são apresentadas considerações finais sobre o trabalho, bem como a bibliografia utilizada.

2 SIMULAÇÃO DISTRIBUÍDA MULTINÍVEL

A simulação distribuída multinível é uma tentativa de acelerar o tempo de simulação de grandes sistemas digitais. Esta técnica combina duas outras técnicas: a simulação multinível e o processamento distribuído.

A descrição de um sistema digital em um só nível de abstração acarreta alguns problemas à simulação. Se o nível de abstração for muito detalhado, próximo ao nível de transistores, a simulação é muito lenta, embora os resultados sejam bastante precisos. Já um circuito descrito em um nível alto de abstração, como nível de sistema, é simulado em um tempo muito menor, porém não se pode obter muitos detalhes do circuito a partir dos resultados da simulação.

A simulação multinível é a técnica que permite a simulação de um circuito digital dividido em diferentes níveis de abstração. Esta técnica permite que partes mais críticas de um sistema digital sejam descritas em níveis de abstração mais baixos e simulados com mais rigor. Outras partes, menos críticas, podem ser descritas em níveis de abstração mais altos e assim simulados. Isto permite um aumento da velocidade da simulação.

Mesmo usando a técnica de simulação multinível, o tempo de simulação ainda é muito alto para sistemas muito grandes.

Outra técnica para acelerar a simulação é o uso de recursos computacionais para processamento distribuído. Pode-se agir de duas formas: da primeira, divide-se o algoritmo de simulação em partes que são executadas em processadores diferentes. Da segunda forma, pode-se dividir o sistema digital em diversas partes, que serão simuladas em processadores diferentes, pelo mesmo algoritmo.

Indo mais longe ainda na tentativa de acelerar o tempo de simulação de grandes circuitos, pode-se, enfim, mesclar as duas técnicas acima, obtendo a simulação distribuída multinível.

Para a construção de um simulador distribuído multinível, é necessário que sejam atendidos alguns requisitos que garantam que o simulador seja realmente distribuído e multinível. Além desses, alguns outros requisitos devem ser observados, a fim de preservar a integridade da simulação. Esses requisitos são:

- descrição de partes diversas do circuito em níveis de abstração diferentes;
- a simulação deve ser executada por processos independentes, em processadores distintos;
- a comunicação entre os processos componentes deve ser feita apenas por troca de mensagens, sem compartilhamento de memória;

- qualquer controle que haja entre os processos deve ser descentralizado, isto é, não deve haver um processo centralizador das ações de controle;

- o tempo de simulação deve ser controlado de forma distribuída, permitindo a sincronização entre os processos.

Entre os itens anteriores, o último foi objeto de um estudo mais aprofundado [FIG92], visando conhecer as técnicas existentes para sincronização de processos em sistemas distribuídos e, conseqüentemente, que pudessem ser usadas na simulação distribuída.

Há dois tipos principais de mecanismos de sincronização descritos na literatura [JEF87] [FUJ90]: mecanismos conservativos, nos quais é mantida a sincronização total entre os vários módulos através de um mecanismo central de tempo, e mecanismos otimistas, baseados em técnicas de tempo virtual, onde cada módulo avança seu tempo independentemente dos outros módulos, havendo, no entanto, um controle de quebra de sincronismo, quando então é feita a retroação do processamento. Esses métodos serão detalhados a seguir.

2.1 Técnicas de sincronização em sistemas distribuídos

2.1.1 Técnicas conservativas

Os primeiros mecanismos de sincronização eram baseados em técnicas conservativas. Exemplos disso são os mecanismos como monitores, *rendezvous*, caixas-postais ou semáforos. Por esses mecanismos, para que os processos se mantenham sincronizados, em algum momento um tem de esperar pelo outro, para, daí, continuarem sincronizados.

Basicamente, as técnicas conservativas se preocupam em evitar qualquer discrepância entre os tempos de execução dos eventos dos processos constituintes do sistema. Um determinado evento E1, com um determinado tempo T1, somente poderá ser executado se houver a certeza de que nenhum outro evento com tempo menor que T1 aparecerá mais tarde para ser executado. Se há alguma dúvida desta regra ser desrespeitada, o processo deverá permanecer bloqueado, aguardando o tempo certo de executar seus eventos.

Como conseqüência desses procedimentos, dois problemas surgem: o primeiro é a determinação correta de que um evento está seguro para ser executado; o segundo, é a grande possibilidade de *deadlocks* nos bloqueios dos processos.

A técnica usada para determinação da exeqüibilidade de um evento é a colocação, em cada ponto de entrada de mensagens de um processo, de uma fila de mensagens de chegada. Cada mensagem, além do seu conteúdo, carrega o tempo no qual deve ser executado o evento (*timestamp*). Portanto, cada ponto de entrada de mensagens em um processo contém uma fila de mensagens e cada mensagem carrega seu *timestamp*. Dentro de cada fila, as mensagens são ordenadas por ordem crescente de *timestamp*. Outra peculiariedade é o fato das filas, mesmo vazias, manterem o *timestamp* da última mensagem retirada. O processo, ao buscar nas filas um evento para executar, sempre busca o de *timestamp* mais baixo. Com isso, se todas as filas tiverem pelo menos uma mensagem, está garantido que cada processo executará sempre os eventos na ordem crescente de tempo e também enviará mensagens respeitando esta regra. Dessa forma, todos os processos componentes do sistema trocarão mensagens respeitando a ordem crescente de tempo, formando, então uma ordenação global.

Se um processo, ao buscar entre suas filas de mensagens o próximo evento a executar, constatar que alguma das filas está vazia e que exatamente esta contém o tempo menor, então o processo deverá ficar bloqueado até que alguma mensagem, chegando pela fila vazia, libere-o do bloqueio.

O maior problema deste método é a grande possibilidade de ocorrência de *deadlock*. Suponhamos que os processos A, B e C comuniquem-se entre si formando um ciclo, de forma que A manda

mensagens a B e recebe de C, B envia mensagens a C e recebe de A e C recebe de B e envia para A. Se as filas de entrada que comunicam os três processos tiverem o menor tempo e estiverem vazias, o três processos ficam bloqueados, configurando uma situação de *deadlock*. A figura 2.1 ilustra a situação.

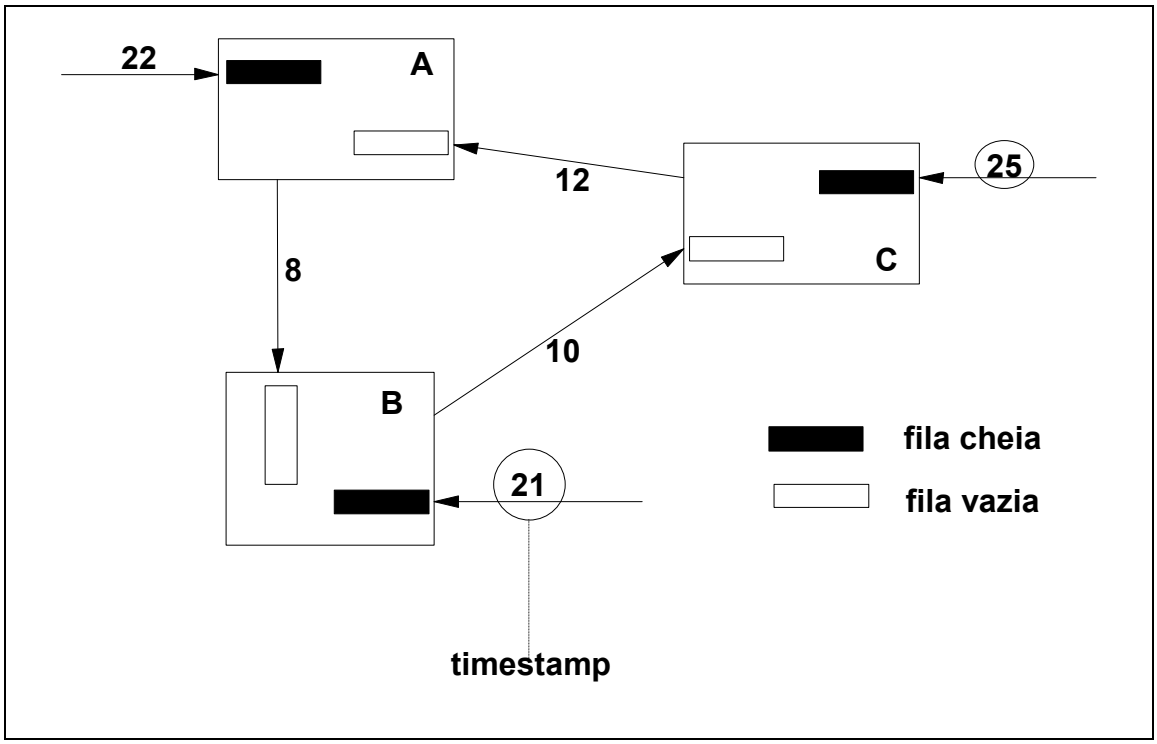

Figura 2.1: Representação de um *deadlock*.

Existem vários métodos descritos na literatura [SU89] [LIU90] para evitar *deadlocks* ou para detectá-los e solucioná-los. Para evitá-los, pode-se usar a técnica de mensagens nulas [CHA79]. Essas mensagens não carregam qualquer informação e têm apenas a função de sincronização. Elas carregam apenas o tempo até o qual certamente não

será enviada nenhuma mensagem válida. Por exemplo, o processo A, após enviar uma mensagem ao processo B com o tempo 8, poderia enviar a B uma mensagem nula com o tempo 25, indicando que até este tempo nenhuma mensagem válida será enviada de A para B. Deste modo, B ficará liberado para buscar eventos a serem processados com tempos menores ou iguais a 25, nas outras filas de entrada.

Este mecanismo realmente evita a ocorrência de *deadlocks*, mas acarreta um grande tráfego de mensagens e um desperdício de tempo de processamento dos processos para controlar internamente o envio dessas mensagens.

Variações desse método são apresentadas na literatura em [CHA81], [LIU90], [MIS86]. Estas versões eliminam o uso de mensagens nulas e permitem que haja *deadlock*. Um mecanismo separado é usado para detectá-lo e outro ainda para interrompê-lo.

2.1.2 Técnicas otimistas

Diferentemente das técnicas conservativas, as técnicas otimistas não evitam que haja discrepâncias de sincronismo entre os processos. Essas técnicas detectam tais problemas e os recuperam.

Assim, os sistemas têm seu processamento em ações atômicas discretas, chamadas transações. Cada subsistema segue seu

processamento independente dos outros e, assim que for detectada uma discrepância no sincronismo dos processos, eles voltam atrás e recomeçam o processamento do início de uma transação segura.

O mecanismo otimista mais conhecido é o chamado *Time Warp*, baseado no paradigma de Tempo Virtual [JEF85] [JEF87]. Por este mecanismo, os erros de causalidade são detectados quando uma mensagem é recebida com um *timestamp* menor do que o atual. Isto significa que esta mensagem deveria ter sido processada há alguns passos de tempo antes, mas não o foi. Neste ponto, os procedimentos de recuperação são acionados.

Esses mecanismos de recuperação baseiam-se em desfazer os eventos que foram realizados desde o tempo em que ocorreu o conflito, até o tempo em que deveria ter sido processada a mensagem atrasada.

Considerando que um processo executa basicamente duas ações, modificação do seu estado interno e envio de mensagens a outros processos, de acordo com o estado atual, essas devem ser contempladas no retrocesso do processamento.

Para retroceder aos estados anteriores, é necessário que o processo salve periodicamente seus estados intermediários. Assim, no retrocesso, basta buscar um estado seguro e restaurá-lo.

Para tratar as mensagens que foram enviadas indevidamente, é necessário enviar mensagens que cancelem o efeito das anteriores. Essas mensagens de cancelamento são chamadas de antimensagens ou de mensagens negativas. As mensagens que não são de cancelamento são chamadas de mensagens positivas.

Se um processo recebe uma antimensagem que corresponde a uma mensagem positiva que já foi processada, ele deve também acionar seu mecanismo de retrocesso, desfazendo suas ações e enviando novas antimensagens. Isto irá ocorrer recursivamente até que todo o sistema esteja em um estado válido.

Nesta técnica de *Time Warp*, o menor *timestamp* entre todas as mensagens (positivas e negativas) não processadas do sistema é chamado de Tempo Virtual Global (TVG). O TVG é considerado o tempo virtual global de todo o sistema e é usado para medir o progresso do sistema, assim como para saber se a atividade do sistema foi completada. Nenhum retrocesso a um tempo inferior ao TVG será efetivado. Assim, os estados com tempo inferior ao TVG, armazenados para fins de retrocesso, podem ser descartados.

As operações de entrada e saída, por sua característica de irreversibilidade, devem ser proteladas até que seu tempo seja menor que o TVG.

Algumas variações desta técnica têm sido propostas pelos pesquisadores. Uma delas analisa o fato de, quando ocorre um conflito provocado por uma mensagem atrasada, nem todas as antimensagens devem ser enviadas [BAE89]. Alguns dos eventos não são afetados pelo conflito e por isso, não precisam gerar antimensagens. Esta técnica, chamada de *lazy cancellation*, não envia imediatamente as antimensagens. Ao contrário, aguarda para ver se o reprocessamento gera as mesmas mensagens. Se as mesmas mensagens forem geradas, não há necessidade de cancelá-las. Uma antimensagem somente é enviada se o tempo local ultrapassar o tempo de envio da mensagem positiva correspondente sem que esta tenha sido gerada.

Muitas outras otimizações vêm sendo propostas na literatura a fim de minimizar os efeitos dos retrocessos e diminuir o tréfego de mensagens entre os processos [AGR91] [BAE89] [LIN90].

A técnica de *Time Warp* tem seus méritos em função da melhor exploração do paralelismo entre os processos integrantes do sistema e da impossibilidade de ocorrência de *deadlocks*. Por outro lado, o tempo e os recursos gastos para processar os retrocessos são fatores que prejudicam seu desempenho. Além disso, a implementação dos recursos necessários para o funcionamento da técnica de *Time Warp* é bem mais complexa que a implementação dos mecanismos conservativos.

Grande parte das publicações recentes sobre simulação distribuída tem se preocupado em melhorar o desempenho da técnica de

Time Warp, seja através de otimizações no seu algoritmo básico, seja pela mescla com mecanismos conservativos.

3 VERSÃO CENTRALIZADA

Todo sistema digital em AMPLO é representado por uma **agência** [WAG88b]. Uma **agência** pode ser descrita de forma puramente estrutural, como uma composição de instâncias de outras **agências**, ao que chamamos **rede de agências**, através de uma linguagem chamada REDES [WAG87b]. A descrição da **agência** pode ser feita alternativamente em um determinado nível de abstração através de construções de uma das linguagens LAÇO, KAPA ou NILO. Uma agência descrita dessa forma é uma **agência primitiva**.

O ambiente de simulação está baseado no princípio mestre-escravo [WAG87c]. Os simuladores operam sobre estruturas de dados que descrevem o sistema digital de forma modular. Um simulador escravo é dedicado a um determinado nível de projeto e interpreta os eventos no interior dos módulos, enquanto que um simulador mestre coordena as interações entre os módulos e é responsável pelo avanço do tempo de simulação. O simulador mestre pode ativar diferentes escravos, de acordo com o nível de descrição do módulo em questão, o que permite a simulação multinível, fundamental no processo de refinamentos sucessivos, normalmente adotado no projeto de sistemas digitais complexos.

Os níveis de abstração nos quais o circuito pode ser descrito são:

- Nível de sistema - permite a descrição do *timing* e do comportamento funcional do sistema digital, sem especificação de quaisquer detalhes de implementação. Este nível é suportado pela linguagem LAÇO [SIL88] [LEF89], baseada em redes de Petri.

- Nível de transferência entre registradores (RT) - permite a descrição do sistema digital em termos de transferência de dados entre registradores. A linguagem para este nível chama-se KAPA [WAG87].

- Nível de portas lógicas - permite a descrição em nível de portas lógicas elementares e chaves bidirecionais. A linguagem associada é chamada NILO [WAG87a].

O ambiente de simulação do AMPLO contém, portanto, os seguintes simuladores:

- três simuladores escravos, um para cada nível de descrição comportamental (LAÇO, KAPA e NILO) [SIL88] [WAG87] [WAG87a];

- um simulador mestre multinível para gerenciamento das interações entre as agências [FIG90];

O ambiente de simulação fornece ainda uma interface padrão com o usuário [WAG90] que suporta diversos mecanismos especializados para controle e gerência do processo de simulação e para interação durante as sessões de simulação. Este módulo será referenciado, a partir de agora, apenas como ambiente.

O simulador mestre é composto de três partes principais:

a) Um núcleo central, responsável pelas funções de propagação de valores de sinais entre os módulos e pela administração de uma lista de eventos que implementa o mecanismo de avanço de tempo. Este núcleo é genérico, independente das linguagens de descrição de *hardware*, e viabiliza a integração de novos simuladores escravos;

b) Adaptadores que permitem a troca consistente de sinais entre módulos descritos em linguagens diferentes;

c) Um módulo gerenciador da comunicação mestre/escravo e mestre/ambiente;

3.1 Algoritmo de simulação

O processo de simulação [FIG90] de um sistema digital em AMPLO começa com a preparação de um modelo de simulação. Um modelo de simulação é uma **rede de agências primitivas**, portanto sem

hierarquia. Isto é feito por um módulo do ambiente chamado construtor de modelos. Este módulo permite a obtenção da estrutura de dados necessária ao processo de simulação, a partir da descrição hierárquica do sistema digital armazenada na base de dados.

No próximo passo, o usuário deve especificar qual o estado inicial do modelo e quais estímulos deseja aplicar em suas entradas primárias. Estes procedimentos são realizados através do ambiente, que oferece as ferramentas necessárias.

Ainda antes de iniciar a simulação, e a cada vez que o controle do programa volta ao ambiente, o usuário pode determinar que sejam feitos alguns controles especiais sobre determinados sinais durante a simulação, tais como INJECT, que permite conectar um estímulo a um sinal do modelo; FORCE, que fixa ("grampeia") um valor em um sinal do modelo de simulação; PUT, que atribui um valor a um sinal, que poderá ser mudado pela simulação; entre outros.

Terminada a fase inicial, o simulador mestre é informado sobre qual modelo de simulação deve atuar, com qual estado inicial, quais os estímulos de entrada e quais os controles intermediários que deve fazer. Só então o algoritmo de simulação é iniciado.

O simulador mestre trabalha essencialmente sobre quatro estruturas de dados:

LISTA DE EVENTOS: é uma estrutura de dados onde ficam ordenados por tempo os próximos eventos programados para as agências;

REDE DE AGÊNCIAS PRIMITIVAS: é a mesma estrutura de dados criada pelo construtor de modelos, que contém as informações sobre as agências que devem ser simuladas e as interconexões entre elas;

TABELAS DE SINAIS CONTROLADOS: são todas as tabelas dos sinais que precisam sofrer algum tipo de controle especial do seu valor durante a simulação, como os sinais monitorados ou grampeados;

LISTA DE ENTRADAS PRIMÁRIAS: esta lista contém todas as entradas primárias do modelo com suas respectivas listas de *fanout* e estímulos associados.

Ao receber o comando de inicialização, o simulador recebe do ambiente os seguintes dados iniciais:

- rede de agências primitivas;
- estado inicial do modelo;
- lista de entradas primárias;
- estímulos a serem aplicados nas entradas primárias.

Neste momento, são efetuadas as inicializações das variáveis do modelo e a preparação do simulador para o início do trabalho. Terminada a inicialização, o mestre devolve ao ambiente o controle do programa e espera que este mande a lista de mensagens indicando quais os sinais que devem ser monitorados durante a simulação e também a mensagem indicando o início efetivo da simulação. Quando essas mensagens são recebidas, o simulador mestre primeiro trata as mensagens de controle de variáveis e, após, tem início o algoritmo principal de simulação.

O algoritmo de simulação começa com a programação, na Lista de Eventos, dos eventos provocados pelos estímulos nas entradas primárias. O mestre começa então a incrementar o tempo e a percorrer a Lista de Eventos a cada novo tempo, chamando os escravos para simularem as agências que estiverem programadas para aquele tempo.

Após a simulação de cada agência, o mestre deve verificar que mensagens foram retornadas do escravo e propagar as mudanças nos sinais de interface entre as agências, ou seja, chamar os escravos para simularem as agências cujos sinais de interface variaram pela ação de outras agências. Ao final de cada passo de tempo, o mestre deve consultar as tabelas de sinais controlados e executar as tarefas relacionadas, como monitoração de sinal, colocar valores em variáveis, etc.

O mestre verifica, então, se já foi satisfeita uma condição de parada. A simulação pode parar por um dos três motivos abaixo:

- o tempo de simulação chegou ao fim;
- alguma condição de *breakpoint* foi satisfeita (esta condição é uma expressão booleana envolvendo valores de sinais);
- algum erro grave aconteceu e impediu a continuação da simulação.

Os erros referenciados acima são detectados pelos simuladores escravos e uma mensagem, ao final da simulação de uma agência, é enviada ao mestre. Nem sempre os erros detectados são motivo de parada da simulação. Nesse caso, uma mensagem indicando o erro e o tempo em que ocorreu é colocada na lista de mensagens a ser enviada ao ambiente e a simulação continua normalmente. Caso contrário, o simulador mestre interrompe a simulação e envia uma mensagem ao ambiente indicando o motivo do erro.

Quando a simulação pára, por algum dos motivos acima, é enviado ao ambiente o tempo atual de simulação e uma lista com as mensagens acumuladas durante a simulação. O mestre então recebe do ambiente as mensagens com os novos sinais a controlar e atualiza suas tabelas. Só então ele incrementa o tempo e repete o processo, até que surja uma nova situação que pare a simulação.

A estrutura de controle da simulação mais importante é a Lista de Eventos, que mantém um lista ordenada de tempos, a cada um dos quais está relacionada uma lista de eventos que devem ser executados naquele tempo.

Existem 4 tipos de eventos:

Evento tipo DELAY: quando há um atraso de propagação em algum ponto interno da agência que está sendo simulada, a simulação desta agência só pode continuar após transcorrido o tempo do atraso. Neste caso, o escravo interrompe a simulação e avisa ao mestre que aquela agência deve ser chamada novamente à simulação depois de n unidades de tempo. O mestre, então, coloca mais um nodo referente na lista de eventos no tempo *t_atual* + *n*, identifica o nome da agência e classifica este evento como DELAY. Como para cada agência já existe uma lista de eventos interna com todos os eventos futuros para ela programados, a lista de eventos do mestre contém sempre apenas no máximo um evento de tipo DELAY para cada agência. O escravo comunica ao mestre, portanto, sempre apenas o próximo tempo no qual a agência deve ser reativada.

Evento tipo CLOCK: alguns sinais da interface das agências podem ser definidos como *clock*, que tem características especiais. Por isso, há um evento especial, chamado CLOCK, que trata

especificamente desses sinais. O nodo de um evento do tipo *clock* guarda o nome do sinal de *clock* que deve ser ativado naquele tempo, ao contrário dos eventos tipo DELAY, que guardam o nome da agência a ser chamada. Quando um evento deste tipo é encontrado, o seu nodo é eliminado e é acrescentada uma programação de evento tipo VARIAÇÃO-SINAL-INTERFACE para cada agência ligada ao sinal. Além disso, o novo tempo no qual o sinal de *clock* deve ser ativado é calculado e uma programação de evento tipo *clock* é feita para aquele tempo.

Evento tipo VARIAÇÃO-SINAL-INTERFACE: este tipo de evento é programado para uma determinada agência quando algum sinal na sua interface mudar de valor. Na execução deste evento, o escravo é chamado para atualizar este sinal e simular a agência.

Evento tipo VARIAÇÃO-SINAL-EXTERNO: este tipo de evento é programado para uma determinada entrada primária, quando ocorrer uma variação no seu estímulo de entrada. O tratamento de um evento desse tipo é idêntico ao dado aos eventos do tipo *clock*, incluindo o cálculo do próximo tempo em que deve ser programada nova variação no sinal, a partir do estímulo associado ao sinal. Este estímulo é uma seqüência de eventos, descrita através de uma linguagem especializada no ambiente.

3.2 Comunicação

Durante todo o processo de simulação, o mestre tem de se comunicar tanto com os escravos quanto com o ambiente de simulação.

Esta comunicação, na versão centralizada, é feita por um "sistema de troca de mensagens". No entanto, como esta versão é implementada em um só processador, a "troca de mensagens", na verdade, é a passagem de parâmetros através da chamada recíproca de funções de um módulo a outro.

O ambiente oferece a possibilidade de o usuário determinar que sejam feitos alguns controles especiais sobre alguns sinais durante a simulação, além de consultar valores de sinais e até o salvamento do estado da simulação a um determinado tempo, para restauração posterior.

Todos esses comandos que são oferecidos pelo ambiente devem ser suportados pelo simulador. Para tanto, são usadas as seguintes mensagens:

INJECT: esta mensagem é enviada ao mestre pelo ambiente quando este quer mandar uma lista de sinais, com os respectivos estímulos, que devem ser aplicados durante a simulação. Junto à mensagem, é passada uma lista de sinal - estímulo;

FORCE: esta mensagem é enviada para indicar uma lista de sinais, com os respectivos valores, que devem ser forçados durante a simulação;

PUT: esta mensagem é enviada ao mestre pelo ambiente para indicar uma lista de valores que devem ser atribuídos a determinados sinais. Note-se que a diferença entre FORCE e PUT é que o primeiro tem prioridade sobre qualquer outra modificação futura que possa acontecer com o sinal, enquanto que o segundo pode ser modificado durante a simulação;

TRACE: envia uma lista de sinais que devem ser monitorados durante a simulação. Isto significa que o comportamento destes sinais deve ser guardados durante toda a simulação e deve ser devolvidos ao ambiente logo que ocorra alguma parada;

SHORT: envia uma lista de duplas de sinais que devem permanecer em curto-circuito durante a simulação. Sempre que ocorrer a variação de um dos sinais da dupla, o outro também será afetado;

BREAK: esta mensagem é usada pelo ambiente para enviar ao mestre uma lista de condições de parada que devem ser avaliadas a cada passo de tempo. O parâmetro passado é uma lista de sinal-valor, sendo que, se o identificador do sinal for a palavra

reservada Tempo-Simul, o valor significa tempo absoluto de simulação.

Todas essas mensagens são passadas quando o ambiente chama uma rotina do mestre chamada MT_MENSAGEM. O único parâmetro desta rotina é uma lista com as mensagens citadas acima. Cada uma dessas mensagens carrega junto uma lista com os parâmetros necessários como, por exemplo, uma lista de sinal-valor, no caso de FORCE. Ao receber cada uma dessas mensagens, o mestre separa os sinais da lista que pertencem às interfaces das agências, providenciando tabelas específicas que serão consultadas por ele durante a simulação. Os outros sinais, que pertencerem ao interior das agências primitivas, serão tratados pelos simuladores escravos. O mestre, então, manda para os escravos a lista de sinais restantes. Cada vez que uma nova mensagem é recebida, as tabelas anteriores são substituídas pelas novas. Se alguma dessas mensagens não for enviada, presume-se que continuam valendo as tabelas anteriores. Se alguma das mensagens for passada, mas sua respectiva lista de parâmetros estiver vazia, entende-se que as tabelas atuais devem ser ignoradas e nenhuma outra colocada em seu lugar.

Existem, também, outras duas funções que podem ser usadas pelo ambiente:

READ: é chamada pelo ambiente para consultar o valor de um sinal. A resposta vem imediatamente e é devolvida no próprio retorno

da função. O único parâmetro passado deve ser o nome do sinal.

SAVE: esta função é acionada para que seja guardado o estado atual da simulação, incluindo o valor de todos os sinais. Esta rotina faz com que o mestre mande uma mensagem aos simuladores escravos para que estes devolvam, em um formato próprio, uma estrutura contendo o valor de todos os sinais internos das agências e outros dados necessários para uma posterior recuperação. Depois de os escravos executarem suas tarefas, é a vez do simulador mestre juntar àquela estrutura a parte que lhe diz respeito. Toda esta estrutura é devolvida então ao ambiente, que se encarrega de guardá-la em disco.

Existe ainda uma mensagem especial que se chama SIMULA, que comanda ao mestre o início do processo de simulação. Junto a esta mensagem é passado o tempo máximo de simulação. Esta mensagem é enviada junto com as outras citadas acima, mas é processada por último, para que a simulação somente inicie depois de tratadas todas as outras mensagens.

Depois de terminado um período de simulação, o controle é devolvido ao ambiente. Neste momento, o mestre envia ao ambiente algumas mensagens, que podem ser:

TRACE: devolve ao ambiente a lista de sinais monitorados com o seu comportamento durante o último período de simulação;

BREAK: devolve ao ambiente o índice da condição de *breakpoint* que provocou a parada. Se esta mensagem não for enviada, o ambiente deve entender que a simulação seguiu até o tempo limite indicado.

ERRO: envia ao ambiente uma lista dos erros que aconteceram durante a simulação e um código indicando se a simulação foi prejudicada por algum desses erros.

A comunicação entre o simulador mestre e os escravos é feita através de um sistema de mensagens e de algumas funções específicas, assim como é feito com o ambiente de simulação.

Há duas formas para a comunicação entre mestre e escravo. A primeira delas é uma lista de mensagens relativas a uma agência que vai sendo guardada junto com as outras informações da agência durante o andamento da simulação e é passada ao escravo quando este é chamado para simular a agência. São as seguintes as mensagens tratadas dessa maneira:

VARIE_SINAL_INTERFACE: envia ao escravo um par sinal-valor significando que o sinal (pertencente à interface da agência) deve ter seu valor alterado para o valor dado;

MONITORE_SINAL: envia uma lista de sinais internos àquela agência

que devem ser monitorados durante a simulação da agência;

MUDE_SINAL: indica ao escravo certos valores que devem ser atribuídos a sinais internos da agência ;

FORCE_SINAL: envia uma lista de sinais que devem ser forçados a determinados valores;

VARIE_CLOCK: avisa ao escravo que um determinado *clock* variou.

O tratamento dado pelos escravos às listas de sinais enviadas nas mensagens FORCE_SINAL, MONITORE_SINAL e MUDE_SINAL é o mesmo dado pelo mestre para as mensagens recebidas do ambiente. Ao receber uma lista de sinais, o escravo os trata até que uma lista nova venha substituir a antiga ou até que uma lista nula indique que o tratamento deve ser cancelado.

As mensagens VARIE_SINAL_INTERFACE e VARIE_CLOCK, ao contrário das demais, só carregam um parâmetro de cada vez, mas cada lista de mensagens pode conter várias dessas mensagens.

O sistema de comunicação também contém as mensagens que o escravo envia para o mestre:

VARIEI_CLOCK: indica ao mestre que um determinado sinal de *clock* na interface da agência teve seu valor alterado;

VARIEI_SINAL_INTERFACE: avisa ao mestre que um sinal, diferente de *clock*, mudou seu valor na interface da agência;

PROGRAMEI_EVENTO: avisa ao mestre que um evento deve ser programado para esta agência em um determinado tempo futuro;

DEVOLVO_MONITORADOS: devolve ao mestre o comportamento do sinais que foram monitorados durante a última simulação;

ERRO: manda ao mestre uma lista dos erros que foram detectados durante a simulação da agência. Junto com cada erro, vem um código indicando a gravidade do erro, isto é, se a simulação deve parar ou não devido ao erro.

Existem ainda três mensagens que são implementadas na forma de funções, pois requerem uma resposta imediata:

INICIALIZE_AGÊNCIA: com esta mensagem, o mestre pede ao escravo que inicialize uma determinada agência com um estado inicial e prepare a estrutura de dados necessária para a posterior simulação;

LEIA_SINAL: o mestre pede ao escravo que devolva o valor de um determinado sinal interno de uma dada agência;

SALVA_ESTADO: com esta mensagem, o escravo é avisado para guardar em uma estrutura de dados todos os valores dos sinais internos da agência, bem como todos os dados necessários para uma futura restauração do estado atual da simulação. Esta estrutura é enviada ao mestre, que agrega suas informações e devolve tudo ao ambiente, em resposta à mensagem SAVE;

3.3 Adaptadores

Como já foi dito anteriormente, o simulador mestre de AMPLO tem como função prover a interação entre as as várias agências pertencentes ao modelo que está sendo simulado. Considerando, no entanto, que essas agências possam estar definidas cada uma em um nível diferente de abstração e que, em cada nível desses, pode haver tipos de sinais bem diversos, a tarefa de propagar sinais entre as agências torna-se bastante complexa.

Para evidenciar isto, pode-se tomar como exemplo a ligação entre sinais pertencentes às linguagens hoje existentes. A figura 3.1 mostra os tipos de dados disponíveis para serem usados nos sinais de interface das agências.

Nível	Tipo de dado	Largura (bits)	Observações
NILO	terminal	n >= 1	
	bus	n >= 1	upbus, downbus, tribus (*)
	clock	uma fase	só para sinais de saída

KAPA	terminal bus clock	n >= 1 n >= 1 n fases	 upbus, downbus, tribus (*)
LAÇO	terminal variable bus control	n >= 1 n >= 1 n >= 1	integer, boolean (**) (***) integer, boolean (**) (***) integer, boolean (**) (***) largura fixa em 1

* qualificador.

** tipo de valor do sinal.

*** largura só é especificada se tipo de valor for boolean.

Figura 3.1: Tipos de dados disponíveis nas linguagens do AMPLO

Se for tomada como exemplo uma ligação de um sinal terminal de LAÇO com outro terminal de KAPA, não há qualquer problema, desde que verificada uma regra básica de consistência que diz que ambos os sinais devem ter a mesma largura em bits.

Por outro lado, se for ligado um sinal terminal de NILO com um sinal control de LAÇO, a conversão de um sinal passado de um para outro não é trivial, pois deve-se levar em conta que os terminais em NILO podem assumir os valores 0, 1 ou X e que não tem sentido a chegada de um valor X em um sinal *control* de LAÇO, que modela um lugar de uma rede de Petry.

Para resolver estes problemas e permitir que o mestre possa ser implementado abstraindo o mais possível as formas internas de tratamento dos sinais de cada linguagem, foi criada uma entidade chamada **adaptador**. Um adaptador tem por função prover uma interface entre o simulador mestre e os escravos. Ele deve passar os sinais que chegam a um escravo para uma forma que este entenda, possibilitando, com isso, que o simulador mestre não tenha que se preocupar com

detalhes de conversão de sinais entre agências. Dentro do mesmo espírito, quando um escravo acaba de simular uma agência e envia mensagens ao mestre, é função do adaptador também fazer com que essas mensagens cheguem ao mestre em uma forma legível.

Os adaptadores são módulos que acompanham os simuladores escravos sendo, portanto, de responsabilidade de quem implementa um simulador escravo fornecer um adaptador que receba sinais de todos os níveis e os converta para o seu escravo e que traduza as formas internas de representação do escravo para o mestre.

3.4 Barramento

Quando um sinal varia na interface de uma agência, o mestre deve propagar esta variação aos sinais que estão ligados com este que variou. Ocorre que, quando esse sinal é um barramento, deve ser feito um consenso entre os valores de todos os seus sinais acionadores para saber o valor final do barramento.

Um problema surge quando se verifica que cada linguagem tem uma maneira diferente de tratar os barramentos. Em NILO, por exemplo, o barramento pode assumir três valores lógicos (1, 0 ou X), além de conter uma intensidade (E, D, W ou Z), trabalhando, portanto, com uma lógica de 12 valores. Já um barramento em LAÇO e KAPA não tem intensidade, podendo assumir somente os valores 0, 1 ou X.

Um sinal *bus*, ao passar de uma agência NILO para uma LAÇO ou KAPA, certamente terá que perder alguma informação.

Esses problemas de conversão de tipos também são muito específicos das linguagens, assim como os problemas tratados pelos adaptadores. A diferença está no fato de que não há como passar a responsabilidade de tratar os barramentos aos adaptadores, uma vez que somente o mestre tem conhecimento da interconexão entre as agências.

Para solucionar este problema foi criado um módulo que, apesar de fazer parte do código do simulador mestre, foi projetado para ser modificado à medida que novos simuladores escravos forem sendo anexados ao sistema.

A função de consenso de barramento que está implementada prevê a ligação apenas de agências tipo laço, KAPA ou NILO. Nesse caso, a solução que foi encontrada consiste em o mestre sempre converter os barramentos envolvidos para a lógica de 12 valores de NILO, aplicar a função de consenso e então converter o valor final novamente para a lógica cada um dos escravos.

4 ESTUDO DAS PLATAFORMAS DE DISTRIBUIÇÃO DISPONÍVEIS

Para a implementação de uma aplicação distribuída, é necessário o uso de mecanismos que permitam a comunicação dos processos componentes do sistema de forma a não ferir as regras básicas que definem um sistema distribuído, isto é, um sistema onde processos se comunicam exclusivamente através da troca de mensagens, não compartilhando variáveis, e onde o tempo de transmissão das mensagens possa ser considerado insignificante, frente ao tempo do processamento dos eventos. Além disso, deve haver mecanismos que permitam o controle do sincronismo dos processos descentralizados, ou seja, sem que haja a necessidade de processos centrais para fazer o roteamento das mensagens ou o direcionamento das operações para outros processos.

Entre as várias opções de recursos para comunicação entre processos remotos existentes, foram escolhidas três (RPC, CPS e HetNOS) para um estudo mais aprofundado, a fim de selecionar uma que possa servir de plataforma para o desenvolvimento de uma versão distribuída dos simuladores do AMPLO.

A seguir, é feito um estudo dessas plataformas e, adiante, é feita a escolha de uma para ser utilizada no projeto.

4.1 Remote Procedure Call

Chamada Remota de Procedimento, ou **RPC** (do inglês Remote Procedure Call), é um mecanismo para comunicação em alto nível entre processos [SUN90]. Este mecanismo permite que as aplicações distribuídas possam ser desenvolvidas abstraindo-se muitos detalhes dos níveis mais baixos de comunicação.

RPC implementa um sistema de comunicação do tipo cliente/servidor. O processo cliente faz uma chamada a um procedimento que está no processo servidor. Quando recebe a chamada, o servidor executa o serviço solicitado e envia de volta ao cliente um resultado.

Localmente em um programa, a chamada de um procedimento provoca a passagem do controle a este procedimento, bem como a passagem de dados como parâmetros. Da mesma forma, uma chamada remota a procedimento faz com que o chamador fique bloqueado, enquanto o servidor é acordado para executar o serviço. Após retornar um valor de resultado ao cliente, o servidor volta a dormir. Nesse momento, então, o cliente volta ao seu processamento.

O objetivo de se usar este mecanismo é fazer com que a comunicação entre os processos seja transparente ao programador. Para comunicar-se com outro processo, um cliente deve apenas chamar um procedimento remoto, da mesma forma que chama um procedimento local.

A figura 4.1 mostra um esquema do funcionamento de uma chamada remota a procedimento.

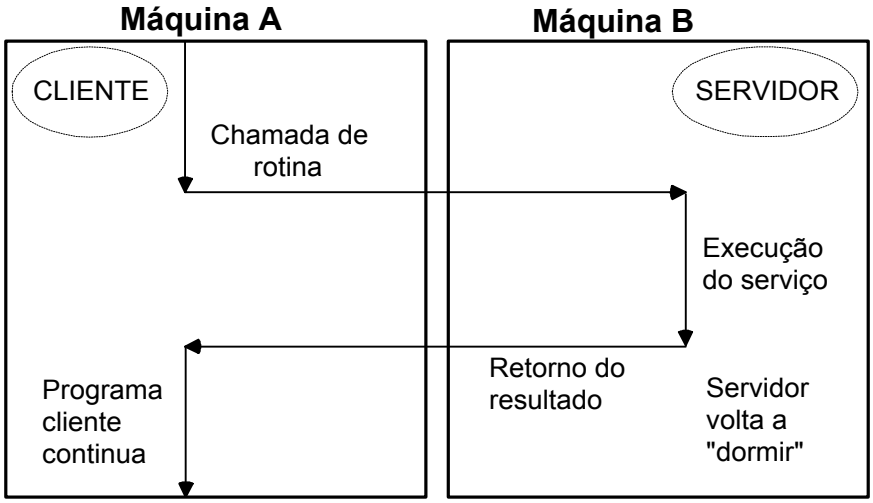

Figura 4.1: Chamada Remota a Procedimento

A comunicação entre os processos é feita com a passagem dos parâmetros do cliente ao servidor e da passagem do resultado do servidor ao cliente.

No modelo RPC padrão, o servidor está “dormindo“ enquanto o cliente processa, e este fica bloqueado enquanto o servidor executa a tarefa solicitada. Isto significa que não existe paralelismo entre eles.

No entanto, diferentes implementações de RPC podem não ser assim. As chamadas podem ser assíncronas, de modo que o cliente não fique bloqueado esperando resposta e, assim, continue seu processamento em paralelo. Da mesma forma, pode-se fazer com que o

servidor dispare uma tarefa (*task*) para executar o serviço solicitado, ficando liberado para atender a outras requisições.

4.1.1 A implementação no SunOS

Cada procedimento remoto é univocamente identificado no sistema através de um número dado ao procedimento e de um número e de uma versão atribuídos ao programa ao qual ele pertence.

Para facilitar a programação, o sistema fornece um utilitário chamado **rpcgen** que auxilia a confecção de programas.

O **rpcgen** nada mais é que um compilador que recebe uma definição de interface do programa remoto, descrito em uma linguagem chamada Linguagem de RPC (similar à linguagem "C"), contendo os números de identificação do programa, da sua versão e dos procedimentos remotos que contém. Ele, então, produz algumas saídas em linguagem "C", que são:

- um esqueleto de cliente;
- um esqueleto de servidor;
- rotinas de transformação dos dados para formato de transporte, tanto para os parâmetros, quanto para o resultado;
- um arquivo com as definições em comum.

O programador, após desenvolver o servidor em alguma linguagem que observe as convenções de chamadas do sistema (como 'C'), deve ligar este código com o esqueleto gerado por **rpcgen** para gerar um servidor executável. Da mesma forma, um programa que use os recursos desse servidor deve ser escrito e depois ligado ao esqueleto, produzindo um arquivo executável.

A figura 4.2 mostra um esquema do uso de **rpcgen**.

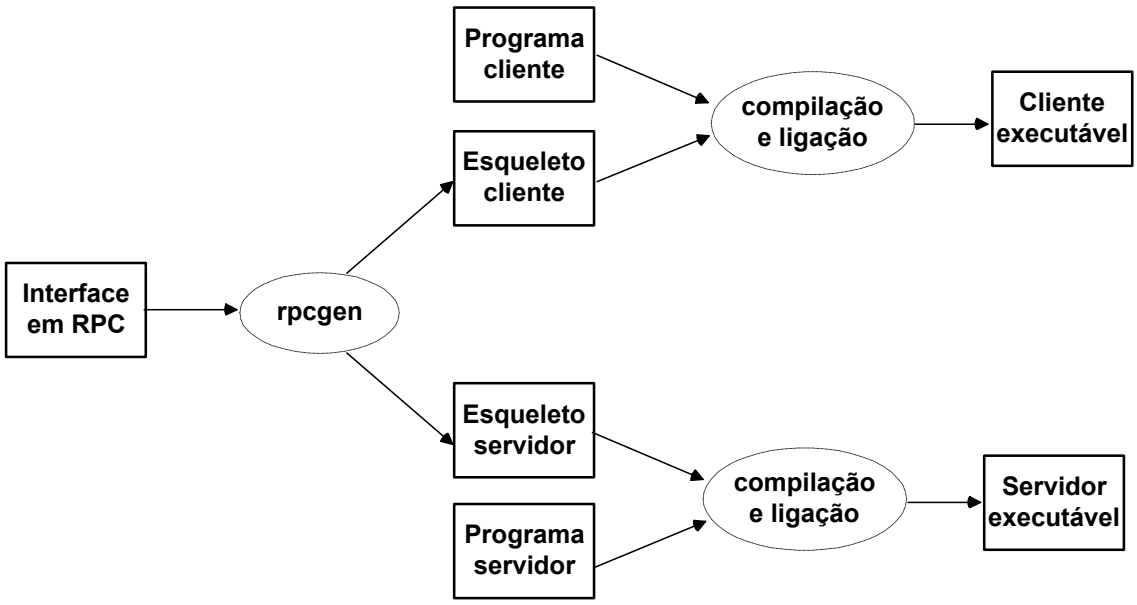

Figura 4.2: Esquema do uso de **rpcgen**.

4.2 Cooperative Processes Software

Cooperative Processes Software (ou simplesmente **CPS**) é um conjunto de ferramentas para o desenvolvimento de software paralelo para a máquina ACP II. ACP II é uma máquina multiprocessadora

utilizada com grande sucesso em centros de pesquisa em física de altas energias [ACP89] [GAR91].

CPS foi projetado para ser executado não só em ACP II, mas também em uma grande variedade de máquinas comercialmente disponíveis, incluindo estações de trabalho com sistema operacional UNIX.

O CPS é constituído de dois blocos distintos:

Job Manager: funciona como um servidor que executa um conjunto de funções especiais em tempo de execução, tais como início e término de todos os processos do sistema, manutenção de um arquivo de segurança (*log file*), manipulação de operações de entrada e saída com terminal, tratamento de exceções, serviços de sincronização de processos, transmissão de mensagens, gerenciamento de filas de processos, etc. Para tanto, o Job Manager utiliza-se das informações contidas no arquivo chamado **Job Description File** (JDF), que contém as informações sobre as classes, os processos e os processadores a serem alocados.

Primitivas CPS: conjunto de rotinas que permitem ao usuário utilizar o ambiente distribuído. Essas rotinas são utilizadas dentro de uma linguagem de alto nível, geralmente "C" ou "FORTRAN". As rotinas CPS, referenciadas dentro de um programa fonte,

são ligadas ao final da compilação ao programa objeto. Assim sendo, durante o desenvolvimento do programa, o usuário pode dispor de todos os recursos oferecidos pelo sistema, como depuradores e utilitários.

A criação de um sistema distribuído usando CPS deve estar estruturada segundo a abordagem de processos cooperantes, isto é, processos que produzem resultados que são recebidos e processados por outros processos, e assim sucessivamente.

Para entender essa abordagem, deve-se, primeiramente, entender alguns conceitos básicos:

Job: é todo conjunto de tarefas a serem realizadas para a solução de um problema. Para cada tarefa dentro de um **job**, existirá um programa e uma classe.

Programa: é o código fonte correspondente à execução de uma tarefa.

Processo: é uma instância de programa em execução com o seu correspondente conjunto de dados.

Classe: é o conjunto de processos idênticos, isto é, processos que executam a mesma tarefa sobre dados distintos. Esse conjunto de processos deve estar sendo executado em processadores de um mesmo tipo. Os processos de uma classe podem

compartilhar um mesmo processador, com outros processos de uma mesma classe, ou de outra classe qualquer.

Note-se que um Job contém Classes. Cada Classe contém vários processos. Todos os processos dentro de uma classe têm o mesmo programa. Processos de classes diferentes têm programas diferentes.

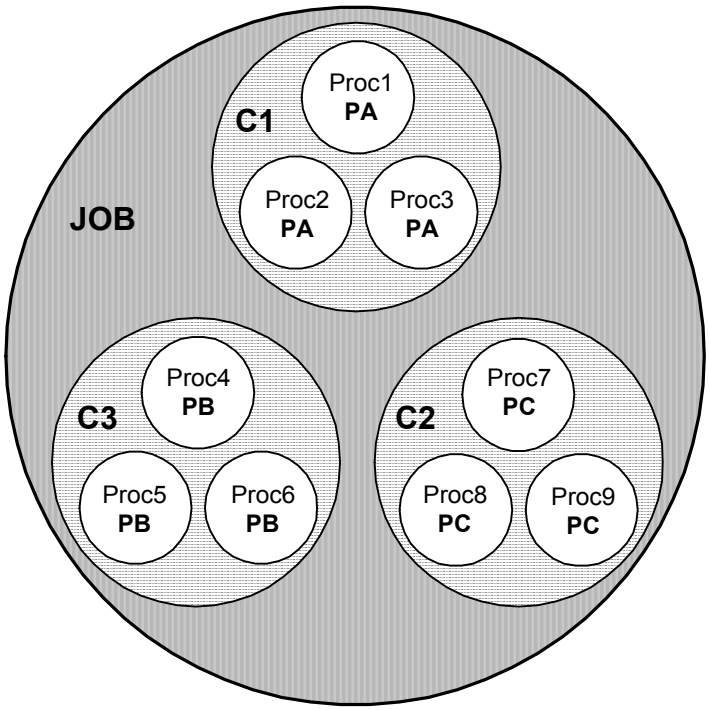

Figura 4.3: Composição de um **job** em CPS.

A figura 4.3 mostra um esquema da composição de um Job em tempo de execução. Nessa figura, C1 é uma classe e, portanto, só contém processos com o mesmo código fonte PA. O mesmo ocorre nas outras classes. O que diferencia os processos dentro de uma classe é o conjunto de dados sobre os quais se dá o processamento. Cada processo atua sobre um conjunto de dados diferente.

Como exemplo de processamento usando esse tipo de abordagem, a figura 4.4 mostra um esquema de um sistema hipotético implementado em CPS.

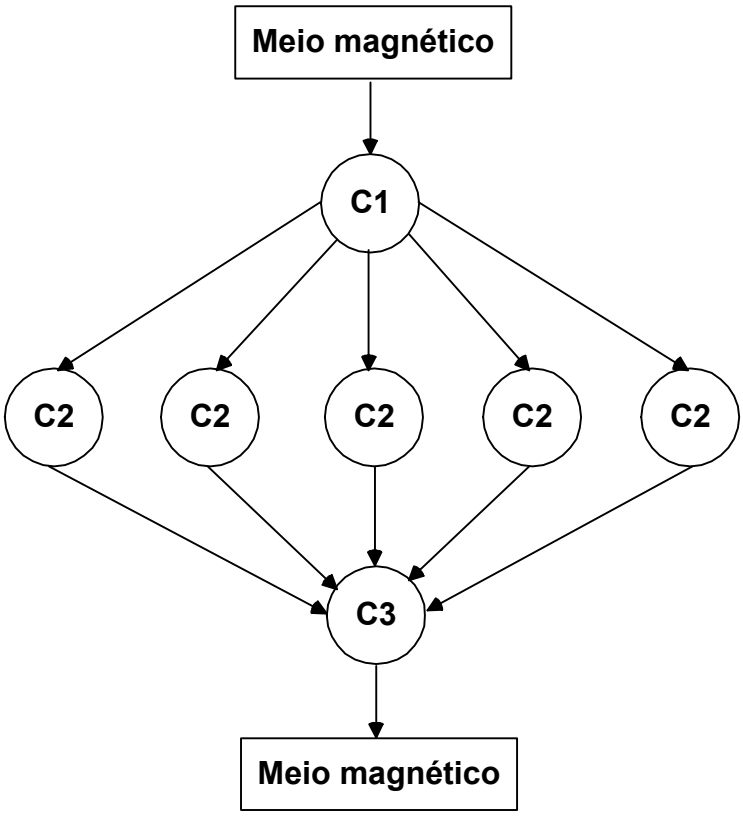

Figura 4.4: Esquema de um sistema em CPS.

Nesse caso são criadas três classes de processos:

- **Classe 1**: nessa classe, o processamento consiste apenas em buscar dados de meio de armazenamento (por exemplo, disco) e passá-lo para um processo da Classe 2;

- **Classe 2**: processos dessa classe procedem a transformação sobre os dados enviados pela Classe 1, através de uma série de cálculos, passando os resultados para algum processo da Classe 3;
- **Classe 3**: aqui os processos estão encarregados de receber os resultados obtidos pela Classe 2 e armazená-los em um meio magnético.

O conjunto de rotinas da biblioteca CPS proporciona vários recursos que facilitam a construção de programas distribuídos, como:

Início e término de programas: rotinas que garantem o início e o término de todos os processos de um Job. Além disso, são acionados mecanismos de controle de estado de cada processo.

Chamada remota de subrotinas: rotinas que permitem que um processo execute uma subrotina de outro processo. Para tornar uma rotina disponível remotamente, o servidor deve declará-la como tal. Ao chamar uma subrotina, o processo envia parâmetros ao servidor e recebe deste argumentos de retorno no final da execução. Enquanto o servidor executa o serviço solicitado, o cliente pode ficar esperando o seu final, pode continuar o processamento independentemente do término da tarefa ou ainda determinar que o servidor seja colocado em uma fila de processos no final da execução da rotina.

Manipulação de filas de processos: rotinas que permitem a inclusão ou retirada de processos em filas de processos. Essas filas são mecanismos criados pelo programador para ter maior controle sobre o estado dos processos. Cada fila é associada a um estado definido pelo usuário. Um processo pode ser incluído ou retirado de uma fila por qualquer processo, inclusive por si mesmo.

Transferência de blocos de dados: rotinas que permitem que um processo faça acesso a blocos de memória de qualquer tamanho de outro. Basta que um processo declare um bloco de dados disponível, para que qualquer outro processo possa lê-lo ou atualizá-lo.

Sincronização de processos: rotinas que permitem a definição de pontos de sincronismo explícito entre os processos de uma ou de diversas classes. O sincronismo é implementado através do conceito de barreira. O processo que executar a rotina de sincronismo ficará suspenso até que todos os processos participantes do bloqueio também executem a mesma rotina.

Transmissão explícita de mensagens: conjunto de rotinas que permitem a transmissão explícita de mensagens entre os processos. Com essas rotinas, o programador pode implementar técnicas de comunicação diferentes das oferecidas por CPS, definindo, inclusive, tipos diferentes de mensagens.

Tratamento de erros e rotinas de uso geral: rotinas que fornecem informações sobre os processos ou classes, ou executam tratamento de exceções.

A preparação e execução de um Job no CPS consiste, basicamente, nos seguintes passos:

1. Criação e compilação de um programa fonte para cada uma das classes existentes, usando as rotinas CPS;
2. Ligação dos programas objeto obtidos com a biblioteca CPS;
3. Criação do **Job Description File** (JDF), isto é, de um arquivo que especificará ao *Job Manager* a alocação dos recursos, a quantidade de processos em cada classe e as máquinas a serem usadas;
4. Execução do *Job Manager*, que lerá o arquivo JDF e criará os processos para execução.

Uma vez criados os processos que formam a aplicação distribuída, deve ser criado o arquivo JDF contendo informações sobre o número de classes e suas identificações, o nome dos programas executáveis de cada classe, o tipo ou nome dos processadores onde as classes serão executadas e o número de processos que será criado em cada classe.

A execução da aplicação é feita sob o controle do *Job Manager*. De posse das informações contidas no arquivo JDF, ele aloca os processadores, dispara todos os processos do *job*, estabelece a conexão entre os processos e atende às requisições de serviço feitas através das primitivas, inclusive as de término dos processos do *job*.

4.3 HetNOS -- Heterogeneous Network Operating System

HetNOS - Heterogeneous Network Operating System é um sistema operacional de rede heterogêneo orientado à programação de aplicações paralelas e distribuídas [BAR92]. é um projeto iniciado em 1991 na Universidade Federal do Rio Grande do Sul e que está disponível na rede de estações de trabalho SUN.

Através de um núcleo distribuído executado em cada máquina, o sistema fornece um ambiente de alto nível para programação distribuída em uma rede de estações de trabalho.

O ambiente de programação oferecido por HetNOS é uma extensão àquele fornecido pelo sistema operacional nativo, o UNIX. Todo o ambiente UNIX é preservado e pode ser utilizado durante o desenvolvimento das aplicações.

Uma aplicação é divida em módulos, sendo alguns destes módulos executados paralelamente com os demais. Cada módulo é implementado como um processo independente, e a interação entre os módulos ocorre via troca de mensagens. O HetNOS fornece um conjunto de funções específicas para tornar possível esta comunicação.

Um dos aspectos importantes de HetNOS é a transparência de localidade para nomes de processos, ou seja, a identificação de um processo não está associada ao nome de um processador (*host*). Quando um processo **P** envia uma mensagem para um processo **Q**, ele não precisa conhecer o nome ou o endereço do *host* onde **Q** está sendo executado. HetNOS se encarrega de procurar o processo destino e lhe entregar a mensagem. Caso o processo inexista, uma mensagem de erro é retornada na função de envio. A identificação dos processos é feita através de nomes compostos por cadeias de caracteres (*strings*), ao invés de números.

HetNOS oferece um ambiente com quatro níveis diferentes de depuração para os programas. Com isso, o fluxo de mensagens entre os módulos pode ser facilmente acompanhado pelo usuário, sem a necessidade de inclusão no programa de linhas de código adicionais para depuração. Os níveis de depuração permitem que o usuário escolha quais as mensagens que deseja acompanhar na tela.

Para que uma aplicação possa criar novos processos, dentro da mesma sessão de usuário, HetNOS oferece um conjunto de primitivas para criação de novos processos e execução de comandos.

A criação de novos processos e a execução de comandos pode ser feita remotamente em outras máquinas. O usuário pode determinar quais as máquinas nas quais deseja que sejam criados e executados os processos. Pode, também, deixar a tarefa de alocar os processos nas máquinas para HetNOS. O critério de distribuição é simples e procura distribuir um número igual de processos para os processadores envolvidos.

A aplicação do usuário deve ser escrita em linguagem "C", usando as primitivas de HetNOS. Após compilado, o programa deve ser ligado com as bibliotecas de HetNOS. Para executar a aplicação é necessário abrir uma sessão, na qual é informado o *login* e a senha do usuário. Se confirmada a identificação, é acionado o *prompt* do sistema, no qual o usuário poderá executar diversas tarefas, principalmente a de comandar o início da sua aplicação.

HetNOS oferece diversas primitivas de comunicação bloqueantes, não-bloqueantes e *multi-cast*, além de primitivas para gerência de processos. As principais primitivas são mostradas abaixo:

- **Primitivas Básicas de Comunicação por Mensagens**

h_send	envio assíncrono
h_sync_send	envio síncrono
h_receive	recepção síncrona

- **Primitivas de Comunicação Não-Bloqueantes**

h_nonblk_send	envio não-bloqueante
h_nonblk_receive	recepção não-bloqueante
h_nonblk_count_messages	verifica quantas mensagens estão disponíveis na fila

- **Primitivas de Comunicação *Multi-cast***

h_multi_send	envio assíncrono de uma mensagem a um conjunto de processos
h_multi_sync_send	envio síncrono de uma mensagem a um conjunto de processos
h_select	recepção síncrona de uma mensagem que pode ter sido enviada por qualquer um dos processos mencionados

- **Primitivas para Gerência de Processos**

h_fork	duplicação de processo
h_loc_exec	duplicação de processo e execução de um comando local
h_rem_exec	duplicação de processo e execução de um comando remoto
h_exec	duplicação de processo e execução de um comando no melhor host disponível escolhido por HetNOS
h_get_parent_name	obtém o nome do processo pai
h_get_application_info	obtém dados sobre um processo em execução
h_find_application	obtém a localização de um processo na rede
h_wait	espera o término de um processo, local ou remoto
h_kill	aborta a execução de um processo

4.4 Avaliação e escolha de uma plataforma

O uso de RPC tem como ponto a favor o fato de ter uma semântica clara e simples. A chamada de procedimentos é muito usada localmente em programas seqüenciais, sendo que seu uso remoto não apresenta nenhuma dificuldade de entendimento.

Além disso, RPC permite uma grande abstração do funcionamento dos níveis mais baixos de comunicação.

Por outro lado, o RPC padrão apresenta o inconveniente de fornecer apenas comunicação 1 para 1, com bloqueio do chamador durante a espera pelo serviço. Assim sendo, apenas um dos processos, entre cliente e servidor, pode estar ativo num determinado momento. A implementação oferecida na rede de estações de trabalho SUN permite burlar essa restrição, o que facilita o uso de RPC.

Da mesma forma, o usuário é responsável pelo controle dos processos em execução, no sentido de escolher e distribuir os recursos, balanceando a carga entre as máquinas e processos, além de ter de tratar as falhas manualmente.

A gerência de processos oferecida em CPS é bastante precária, pois apesar de executar diversas tarefas, ainda exige muita interferência do usuário.

Por ter sido criado para uma filosofia de processos cooperantes, torna-se difícil e trabalhoso usar CPS em outro tipo de aplicação que não siga estas características.

Os vários tipos de comunicação disponíveis em CPS (chamada remota de subrotinas, mensagens explícitas e compartilhamento de memória) permitem boa flexibilidade para as aplicações, que não se limitam ao uso de um só mecanismo.

Além disso, não são oferecidos mecanismos de balanceamento de carga dos processos nos processadores nem recursos para migração de processos entre as máquinas.

Outro fator importante é a precariedade do mecanismo de sincronização que é oferecido. Outro tipo de sincronização diferente do disponível terá de ser implementado através da troca de mensagens, pelo próprio programador.

Já em HetNOS, a principal vantagem está na transparência de localização dos processos na rede. O usuário não precisa saber em qual processador está sendo executado determinado processo. Para identificá-lo, basta o nome. No entanto, se houver a necessidade de intervir na localização, o usuário poderá facilmente fazê-lo.

Outra grande vantagem de HetNOS está no alto nível das primitivas de comunicação e gerência de processos. Com elas, o usuário fica afastado dos níveis mais baixos de comunicação, podendo preocupar-se mais com sua própria aplicação.

Apesar do mecanismo de balanceamento dos processos nas máquinas ser simples, ele já livra o usuário de ter de fazê-lo.

O fato de HetNOS identificar os processos por seus nomes, e não por números, também traz vantagens para a programação e manipulação dos processos.

Para escolha de uma plataforma para desenvolvimento da versão distribuída dos simuladores do AMPLO, procurava-se um ambiente que oferecesse recursos de alto nível para comunicação, evitando que o usuário tivesse de recorrer a níveis mais baixos de implementação. Além disso, eram desejáveis bons recursos de gerenciamento e monitoração dos processos.

Apesar de ser um projeto recente, nos testes feitos para escolha de uma plataforma, HetNOS mostrou-se bastante eficiente, oferecendo os recursos desejáveis, exigindo do programador poucos esforços na implementação da comunicação, deixando-o mais concentrado na aplicação propriamente dita.

Além disso, o ambiente de execução oferecido por HetNOS não requer maiores conhecimentos da arquitetura onde está sendo executada a aplicação.

A documentação *on-line* e a assistência da equipe de desenvolvimento foram também importantes na escolha desta plataforma.

Portanto, as referências à implementação distribuída dos simuladores serão feitas usando as primitivas de HetNOS.

5 ESTUDO DAS POSSIBILIDADES DE DISTRIBUIÇÃO

Existem várias possibilidades para desenvolver uma versão distribuída dos simuladores do AMPLO, baseada na versão centralizada, mostrada no capítulo anterior.

Duas versões foram escolhidas para serem estudadas com detalhes: a versão **totalmente distribuída** e a versão **parcialmente distribuída**.

A primeira delas prevê a inexistência de qualquer controle central entre os processos. Esta ainda foi dividida novamente em duas outras versões, uma com sincronização otimista e outra com sincronização conservativa.

Já a segunda versão conserva o módulo **mestre**, mas paraleliza os processos dos simuladores **escravos**. Esta versão, por sua característica centralizadora, é implementada com sincronização conservativa.

5.1 Versão totalmente distribuída

A versão totalmente distribuída dos simuladores do AMPLO prevê a independência dos simuladores escravos do controle exercido pelo simulador mestre.

As funções executadas pelo mestre, principalmente a de intermediário na passagem de mensagens entre os escravos e a de responsável pelo avanço do tempo, são absorvidas ou pelos próprios escravos ou pelo módulo ambiente de simulação.

Os escravos não utilizam mais o mestre para trocar mensagens entre si. Cada escravo envia a mensagem desejada diretamente para os módulos aos quais esteja ligado.

O avanço do tempo é feito independentemente por parte de cada escravo, sem a intervenção do mestre.

Embora o mestre seja dispensável nessa versão, ele ainda será utilizado para agrupar as novas funções necessárias para a implementação da versão distribuída. Estas funções poderiam ser implementadas diretamente no ambiente de simulação, mas pode-se facilmente aproveitar o módulo mestre antigo, eliminar as funções que não são mais necessárias e colocar as novas, sem alterar o código do ambiente.

Os processos que simulam as agências primitivas são chamados de processos escravos, embora na versão totalmente distribuída não se submetam a nenhum controle central durante a execução da simulação. O termo “escravo” será usado para manter a compatibilidade com a terminologia usada na versão centralizada.

Alguns aspectos importantes devem ser analisados na passagem da versão centralizada para a versão totalmente distribuída, como é mostrado a seguir:

Conhecimento da topologia: o modelo de simulação a ser usado na versão centralizada era determinado pelo usuário através do ambiente de simulação. Esse, por sua vez, montava a topologia com uma estrutura de dados adequada e a repassava ao mestre. Os escravos não tomavam conhecimento desta topologia. As ligações entre os escravos eram feitas pelo mestre. Como na versão totalmente distribuída os escravos devem enviar suas mensagens diretamente aos módulos vizinhos, cada escravo terá de conter as informações sobre os módulos para os quais deverá enviar os sinais de saída. Isto significa dizer que cada escravo deverá ser informado sobre o nome de cada módulo ligado a cada um dos seus sinais de saída. Para tanto, o mestre deve receber do ambiente a topologia completa do modelo e, ao acionar cada escravo, repassar as informações sobre as agências às quais ele está ligado. Na figura 5.1, é mostrada uma parte de uma rede de agências possível. Note-se que os sinais podem ser compostos e decompostos para formar sinais de largura maior ou menor. A descrição dos sinais de interface de uma agência deve prever esta possibilidade. Na figura 5.2, é mostrada a estrutura de dados que descreve as conexões da agência A. Nessa estrutura estão listados todos os sinais de interface da agência. Para cada sinal, está descrito o nome da agência e do

sinal ao qual ele está ligado. Se o sinal é de entrada, composto por partes de sinais de outras agências, a estrutura também reflete esta situação. Como exemplo disso há o sinal X da agência A que é composto por dois outros sinais das agências B e C.

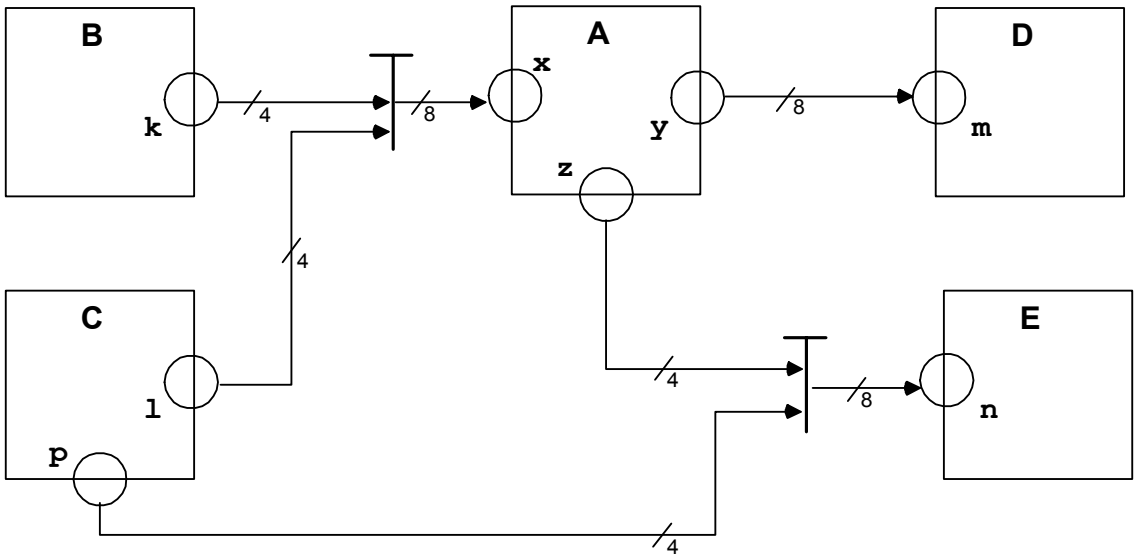

Figura 5.1: Exemplo de uma parte de uma rede de agências.

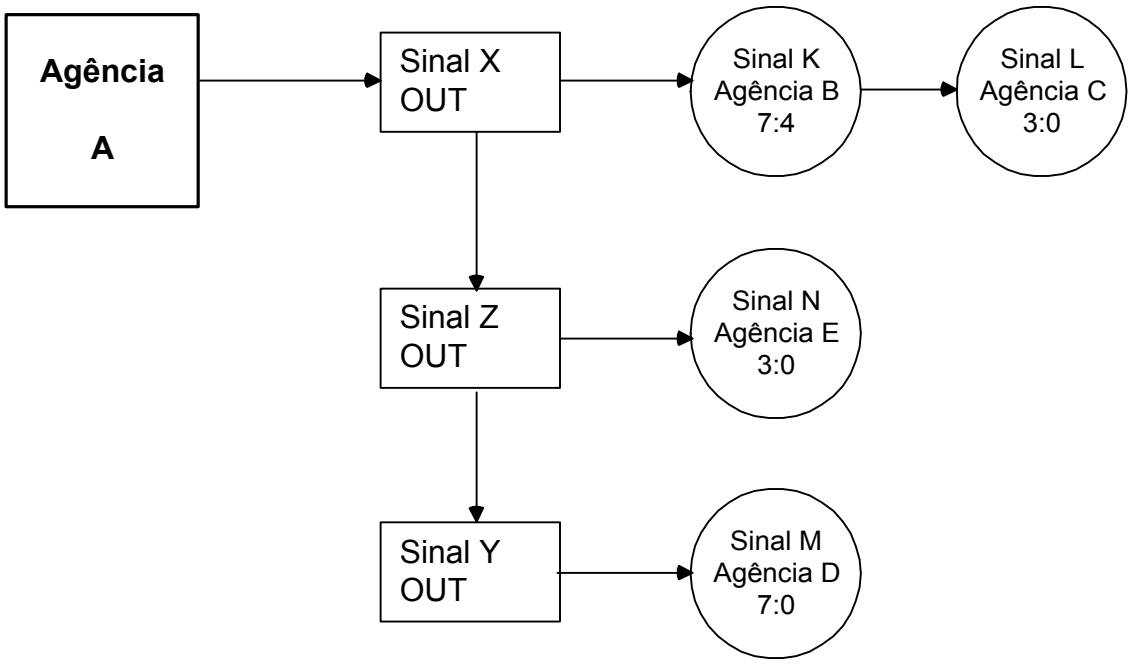

Figura 5.2: Estrutura de dados para descrever as conexões da agência A.

Estímulos iniciais: além da descrição das conexões, o escravo deverá receber a lista de estímulos iniciais que deverão acionar o processo de simulação. Esta lista consiste de formas de onda que serão aplicadas aos sinais. A cada estímulo é associado um identificador e uma lista de trechos de formas de onda, onde cada trecho contém dados sobre a periodicidade, o tempo de duração e uma lista com a descrição do comportamento do estímulo. A figura 5.3 mostra a estrutura de dados que modela a lista de estímulos. Acompanhando a lista de estímulos, o escravo deverá receber também uma lista associando o sinal de interface ao estímulo correspondente.

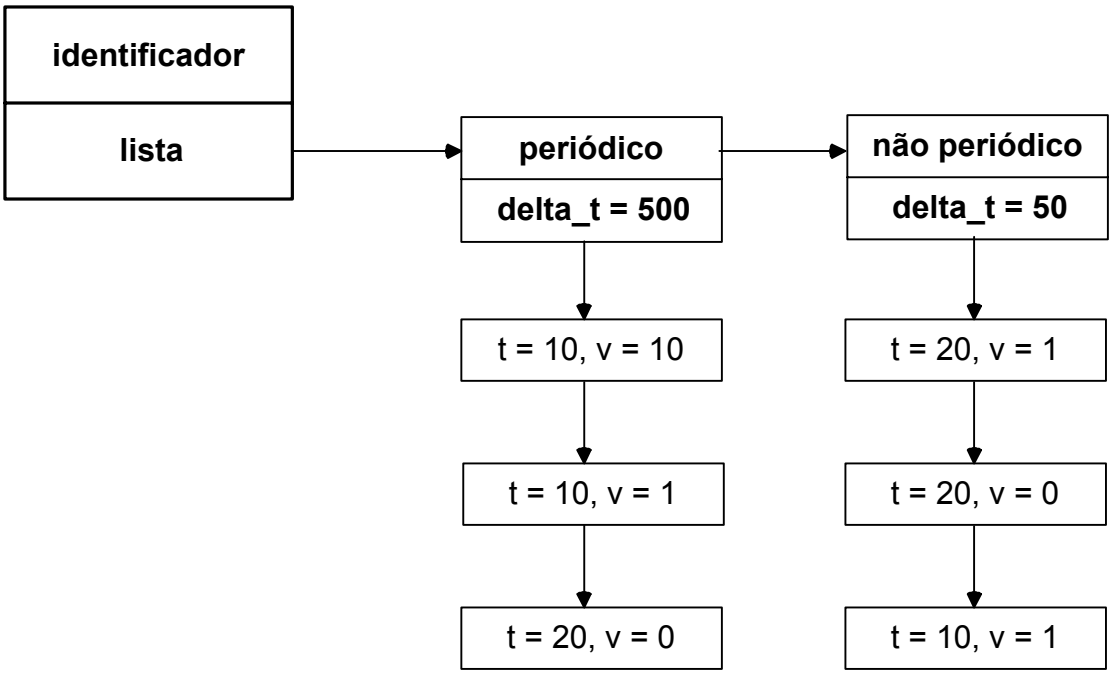

Figura 5.3: Estrutura da lista de estímulos.

Acionamento dos processos escravos: ao receber um modelo de simulação e uma ordem do ambiente para iniciar o processo de simulação desse modelo, o mestre deve percorrer a estrutura de dados que define o modelo e executar, para cada uma das agências primitivas, um processo escravo. Após chamar cada processo, o mestre deve enviar a cada um as informações sobre as conexões daquele processo. Essas informações consistem em identificar cada um dos sinais de saída com o processo que deverá receber aquele sinal. Com essa informação, o escravo será capaz de enviar as mensagens relativas às variações dos sinais de saída diretamente aos processos correspondentes.

Encerramento dos processos escravos: O mestre também é responsável pelo encerramento dos processos escravos ao final da

simulação. O encerramento poderá ser provocado por dois motivos: pela troca do modelo de simulação ou pela finalização da sessão de simulação. Quando há a mudança do modelo de simulação, o mestre recebe o novo modelo do ambiente, encerra os processos escravos anteriores e inicia os novos. Assim, também no encerramento de uma sessão de simulação o mestre procede o término dos processos escravos.

Comunicação dos escravos com o ambiente de simulação: A comunicação dos processos escravos com o ambiente de simulação também é feita através do mestre. Para tanto, o mestre recebe as requisições do ambiente e se comunica com os escravos para obter as informações ou interferir no valor dos sinais. O método usado para troca de informações entre mestre e ambiente na versão centralizada deve ser totalmente modificado para a versão totalmente distribuída. Na versão centralizada, o mestre interagia com o ambiente e escravos cada vez que a simulação parava. Na versão totalmente distribuída, como o mestre não tem mais o controle do tempo de simulação, a comunicação deve acontecer simultaneamente com o processamento da simulação. Para isso, mestre, ambiente e escravos comunicam-se através de mensagens assíncronas, implementando filas de mensagens. A figura 5.4 mostra um esquema do fluxo de mensagens entre ambiente-mestre-escravos.

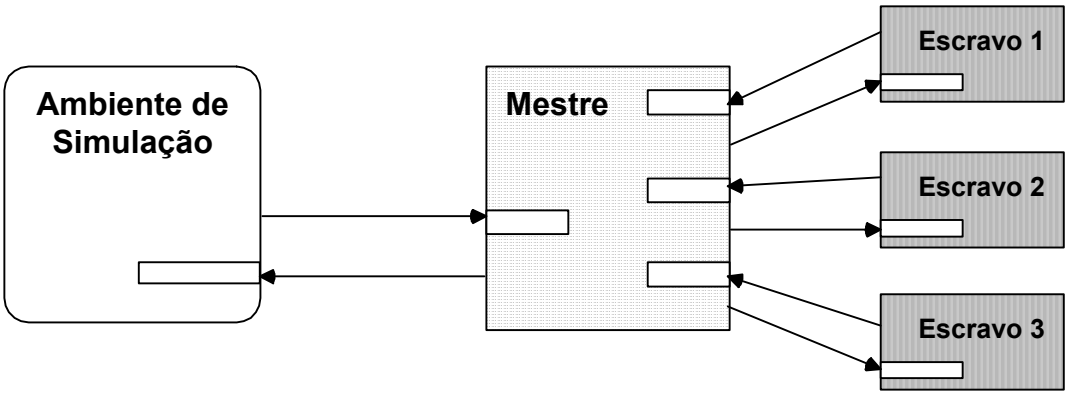

Figura 5.4: Fluxo de mensagens entre ambiente-mestre-escravos.

Barramento: Como foi visto na descrição da versão centralizada, quando ocorre a variação de um sinal de saída que está ligado a um barramento, deve ser estabelecido um consenso entre os valores de todos os sinais acionadores do barramento. Esse consenso era feito pelo mestre na versão centralizada. Na versão totalmente distribuída, há duas maneiras de resolver o problema. Pela primeira, o mestre continua com esta função e, pela segunda, os próprios processos escravos executam o consenso.

A opção do mestre executar o consenso de barramento implica que o mestre mantenha para si a informação de quais sinais compõem os barramentos existentes no sistema. Além disso, cada vez que um sinal do tipo barramento tem seu valor alterado, o processo que originou a alteração envia o novo valor para o mestre, e não para algum outro processo. O mestre executa o algoritmo de consenso e informa a cada um dos processos componentes o resultado. Esta solução não atende a

intenção inicial da versão totalmente distribuída de não haver um controle central do mestre no processamento da simulação.

A outra solução, ou seja, a execução do consenso de barramento pelos próprios processos escravos é de difícil implementação, já que todos os processos acionadores do barramento devem ter conhecimento dos "vizinhos" e trocarem informações entre si para chegar a um consenso. Apesar de ser de mais difícil implementação, esta solução é mais compatível com a intenção de eliminar qualquer centralização na simulação.

Adaptadores: Quanto aos adaptadores, que têm a função de compatibilizar os diversos tipos de sinais entre as agências primitivas de tipos diferentes, pode-se sugerir três soluções distintas.

Pela primeira delas, os adaptadores seriam incorporados em cada um dos processos escravos. Como cada processo tem a informação de qual agência deverá receber as informações de alteração de sinal de saída, ele pode, também, armazenar o tipo de agência primitiva que está ligado a um determinado sinal de saída. Antes de enviar uma mensagem de alteração de um sinal de saída, o processo fonte deve executar a função de adaptação de sinal. Assim, o sinal chegará já na forma correta no destino. Essa função de adaptação também poderia ser implementada para tratar os sinais que entram, e não os que saem. Dessa

forma, um processo enviaria uma mensagem de alteração de sinal sem importar-se com a agência destino. Este sinal seria tratado através de um função de adaptação na entrada da agência destino. Essa solução tem o inconveniente de não permitir a abstração, por parte dos escravos, da existência de outros tipos de agências primitivas além do seu. Os escravos são forçados, para incorporar as rotinas de adaptação, a manter em seu código informações sobre os tipos de sinais das outras agências primitivas e as equivalências entre eles.

Outra solução seria o uso de uma classe de processos adaptadores, de forma que instâncias distintas desta classe seriam criadas cada vez que uma adaptação fosse necessária. Os próprios escravos ativariam um processo adaptador quando tivessem algum sinal a ser enviado a uma agência de outro tipo primitivo. Os processos adaptadores seriam destruídos assim que tivessem cumprido sua tarefa.

A terceira alternativa prevê que, para cada conexão entre agências de dois níveis distintos, seja criado um processo adaptador já na inicialização. Esse processo ficaria à disposição dos processos escravos correspondentes para ser ativado quando do envio de sinais de uma agência a outra.

Uma outra solução seria o uso de um processo auxiliar que executasse a adaptação dos sinais, intermediando o envio dos

sinais de um processo a outro. Essa solução insere um elemento centralizador no processamento do escravos, o que foge à intenção de deixar os processos escravos independentes.

O avanço do tempo de simulação, que na versão centralizada era feito pelo mestre, na versão totalmente distribuída deve ser feito pelos próprios processos escravos. Isto faz com que os processos escravos devam manter um sincronismo entre si. Este sincronismo pode ser implementado de duas formas distintas: com uma sincronização otimista ou com uma sincronização conservativa. As próximas seções analisam os aspectos das duas implementações.

5.1.1 Com sincronização otimista

Quando o mestre dispara os processos escravos para simulação, todos estão no tempo zero. A partir daí cada processo avança o tempo de simulação conforme sua atividade interna. Com o passar do tempo, e de acordo com a diferença de atividade interna de cada processo, podem ocorrer discrepâncias no tempo de simulação entre os processos escravos.

Considere-se que a agência A tem um sinal de saída que está ligado ao sinal de entrada da agência B. Considere-se, também, que, para um determinado tempo real, o processo que simula a agência A está no tempo lógico X e que o processo que simula a agência B está no tempo lógico X+delta. Nesse momento, o processo da agência A detecta uma

variação no sinal de saída que está ligado à agência B e envia uma mensagem ao processo de B comunicando esta alteração. A mensagem é composta do novo valor do sinal e do tempo em que foi gerado aquela alteração. O processo B, ao receber a mensagem de A, constata que o seu tempo é superior ao da mensagem.

Ocorre então um conflito de tempos que deve ser solucionado através do retrocesso das atividades do processo B e do envio de antimensagens para cancelar as mensagens que foram enviadas erroneamente.

Para que seja possível efetuar o retrocesso do processamento e do envio de mensagens, é necessário que os processos escravos mantenham algumas informações, como as que seguem:

- O nome do processo, que deve ser único no sistema;
- O seu tempo virtual local, ou seja, o tempo atual do seu processamento;
- O seu estado atual, que consiste no estado das suas variáveis locais, sua pilha e seu contador de programa.
- Uma tabela de estados, que contém cópias dos seus estados mais recentes;
- Uma fila de mensagens para cada sinal de entrada, que contém as mensagens recém chegadas em ordem de tempo. Ficam nessa fila tanto as mensagens que ainda não foram processadas, como as que já foram processadas, mas seu tempo ainda não é

menor que o TVG e, portanto, ainda podem ter de ser reprocessadas, caso aconteça algum *rollback*;

- Uma fila de mensagens para cada sinal de saída, contendo uma cópia, ordenadas por tempo de envio, das mensagens que foram recentemente enviadas. Estas mensagens são necessárias caso ocorra *rollback* e as mensagens enviadas tenham que ser desfeitas. Apenas as mensagens com tempo superior ao TVG devem permanecer na fila.

Para cada mensagem enviada, existe uma cópia exatamente igual chamada de **antimensagem**. Para diferenciá-las, as mensagens recebem um sinal positivo (**+**) e as antimensagens recebem um sinal negativo (**-**). Toda vez que um processo escravo enviar uma mensagem, esta conterá o sinal **+** e sua cópia irá para a fila de mensagens de saída com o sinal **-**. Assim, o procedimento de enviar uma antimensagem consiste apenas em enviar a mensagem da fila de saída, que já estará com o sinal **-**.

Um exemplo das estruturas mostradas acima está nas tabelas a seguir. Na tabela 5.1, é mostrada a **fila de mensagens de entrada**, com as informações de tempo de envio da mensagem, tempo de recebimento, identificador do processo fonte, identificador do processo destino, sinal (positivo ou negativo) e conteúdo. Analogamente, a **fila de mensagens de saída**, mostrada na tabela 5.2, contém os mesmos campos.

Tabela 5.1: Exemplo de **fila de mensagens de entrada** para um processo escravo.

Tempo de envio	110	105	120	125	130	146	175	176
Tempo de recebimento	112	119	121	141	156	162	181	182
Processo Fonte	E	C	B	E	B	D	C	D
Processo Destino	A	A	A	A	A	A	A	A
Sinal	+	+	+	+	+	+	+	+
Conteúdo	...	...	...	...	...	...	...	...

Tabela 5.2: Exemplo de **fila de mensagens de saída** para um processo escravo.

Tempo de envio	119	121	141	141	156	156	162
Tempo de recebimento	141	122	142	196	180	157	163
Processo Fonte	A	A	A	A	A	A	A
Processo Destino	C	B	D	E	B	C	E
Sinal	-	-	-	-	-	-	-
Conteúdo	...	...	...	...	...	...	...

Devido às peculiaridades de cada simulador escravo, um estudo mais profundo terá de ser feito para determinar as estruturas de dados necessárias para armazenar os dados suficientes para a recuperação. é certo, porém, que o mecanismo de salvamento não precisará guardar todas as informações desde o início da simulação. O tempo mínimo a partir do qual devem ser mantidas as informações é determinado pelo Tempo Virtual Global (TVG). O TVG é o menor tempo lógico entre todos os processos ativos e é o limite abaixo do qual é garantido que não há retrocesso, já que mesmo o processo mais "atrasado" já o superou. Este tempo deve ser calculado pelo mestre, que periodicamente dispara mensagens a todos os processos da rede perguntando por seus tempos lógicos. Os processos, então, calculam o

seu tempo lógico e comunicam ao mestre. De posse dos tempos lógicos de todos os processos, o mestre calcula o tempo virtual global e o envia, finalmente, aos processos.

O armazenamento das mensagens pode ser feito guardando-se, para cada sinal de saída, uma fila de mensagens enviadas. A quantidade de mensagens armazenadas nessa fila e a periodicidade com que serão descartadas também é determinada pelo Tempo Virtual Global. Quanto mais freqüentemente for realizado o cálculo do TVG, melhor será o gerenciamento de memória, visando guardar a menor quantidade de informação possível. No entanto, o cálculo do TVG acarreta uma carga de mensagens bastante considerável no sistema, competindo com o processamento dos eventos da simulação. Portanto, a periodicidade do cálculo do TVG deve ser um compromisso entre quantidade de memória gasta para armazenas os estados passados e a quantidade de mensagens e processamento concorrentes com a simulação.

Assim como acontece com as mensagens trocadas entre os escravos, as mensagens enviadas ao ambiente de simulação também somente poderão ser realmente efetivadas quando seu tempo de envio foi menor que o TVG, já que essas mensagens, assim como procedimentos de entrada e saída, não podem ser retrocedidos.

5.1.2 Com sincronização conservativa

A diferença entre a versão totalmente distribuída com sincronização otimista e com sincronização conservativa está no fato

desta última não necessitar o retrocesso das atividades do processo, já que a sincronização entre os processos sempre será mantida.

Nessa versão, os processos implementam filas para cada ponto de entrada de mensagens. Cada mensagem enviada carrega consigo o tempo lógico do processo que a gerou. Todas as filas de entrada são ordenadas em ordem crescente de tempo das mensagens. Uma fila vazia conserva o tempo da última mensagem que ali esteve. Um processo escravo sempre procurará tratar a mensagem com menor tempo entre todas as mensagens de todas as filas, considerando inclusive as filas vazias. Quando o menor tempo entre todas as filas pertencer a uma fila vazia, o processo permanecerá bloqueado aguardando que, por aquela fila, chegue alguma mensagem que possa ser, então, processada.

Com o uso das filas de mensagens, fica implementada a abordagem conservativa para o sincronismo entre os processos escravos. Os processos sempre estarão sincronizados pelo menor tempo entre eles e será evitado qualquer conflito entre o tempo lógico dos processos e o tempo das mensagens de entrada.

No entanto, esta abordagem pode provocar *deadlocks*, como foi mostrado no item 2.1.1. Para evitá-lo, sugere-se o uso de mensagens nulas. Para cada sinal de saída, são enviadas mensagens nulas que não contêm qualquer informação além de um *timestamp*. Este *timestamp* informa ao processo destino que, até o tempo indicado na mensagem, nenhuma outra mensagem será enviada para aquela entrada.

Como os simuladores escravos foram desenvolvidos usando uma estrutura interna chamada de **lista de eventos**, torna-se fácil implementar tal solução. Esta lista de eventos mantém ordenados, por ordem crescente de tempo, os próximos eventos a serem executados. Assim, basta percorrer esta lista de eventos e encontrar o próximo tempo em que está programado um evento para aquele sinal de saída.

Em termos de sincronização, não ocorre, na abordagem conservativa, nenhum controle por parte do módulo mestre. Toda a sincronização e detecção de *deadlocks* é feita pelos processos.

5.2 Versão parcialmente distribuída

Uma alternativa à distribuição total é a versão parcialmente distribuída, ou versão semi-distribuída, onde o mestre não é apenas um iniciador dos processos. Ele continua aqui com as funções de avançar o tempo de simulação para todos os escravos, de comunicar variações de sinais de interface entre agências vizinhas, de executar os algoritmos de consenso de barramento e de adaptação de sinais, quando necessário.

Nessa versão, o mestre recebe o modelo a ser simulado e cria tantos processos escravos quantos necessários para a simulação de cada agência primitiva. Todos os processos escravos começam com o tempo 0. A partir daí, cada comunicação de um processo com seus vizinhos será

feita através do mestre. Além disso, os processos escravos que não tiverem mais atividades para um determinado tempo, avisarão ao mestre o tempo em que desejarem ser "acordados". Para tanto, o mestre mantém uma estrutura de dados chamada Lista de Eventos, que guarda os próximos eventos de cada agência, ordenados por tempo.

O algoritmo de simulação aqui fica muito semelhante ao algoritmo centralizado. A diferença está na forma de envio e recebimento das mensagens, que aqui são feitas através de primitivas de comunicação entre processos, usando o sistema operacional de rede heterogêneo HetNOS.

Diversos aspectos devem ser analisados, quando da passagem do algoritmo centralizado para o semi-distribuído, como será mostrado nas seções seguintes.

5.2.1 Ambiente de simulação

O ambiente continua fazendo a interface entre o usuário e o simulador mestre. A diferença esta na forma de comunicação, que agora será feita pela troca real de mensagens, já que ambiente, mestre e escravos serão processos distintos, podendo estar ou não sendo executados no mesmo processador.

Com isso, a forma de ativação do mestre, que anteriormente era feita através de uma chamada de subrotina, agora deverá ser feita através

da chamada para execução de um outro processo, usando a primitiva de HetNOS **h_exec**. Com esse comando, o ambiente dispara a execução do mestre, que aguardará a chegada de mensagens.

A figura 5.5 ilustra as mensagens trocadas entre mestre e ambiente de simulação.

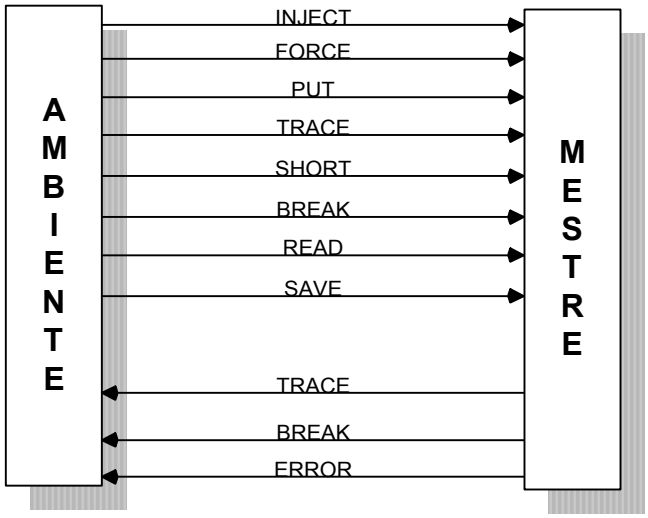

Figura 5.5: Mensagens trocadas entre ambiente e mestre.

5.2.2 Inicialização das agências

Depois da ativação do mestre, a primeira mensagem que ele espera é a que informa o modelo de simulação, o estado inicial e os estímulos a serem aplicados nas entradas primárias. Neste momento, o mestre ativa os escravos nos processadores, de forma que cada agência seja simulada por uma instância do seu simulador respectivo. Isto significa que o mestre percorre a estrutura de dados que contém a

descrição da rede de agências e cria um processo para cada agência existente na rede. Para isso, o mestre usa a primitiva **h_exec** de HetNOS, que permite a criação de um processo remoto sem indicar o processador da rede. A escolha do processador fica a cargo de HetNOS. Os processos serão identificados pelo mesmo nome da agência que simulam. A figura 5.6 mostra um trecho de algoritmo que implementa a ativação de um escravo para cada agência existente na rede de agências primitivas, selecionando o processo escravo pelo tipo da agência primitiva.

```
enquanto agencia.prox != NULL
inicio
    agencia = busca_agencia(rede_de_agencias)
    se agencia.tipo == KAPA
         entao host = h_exec( agencia.nome, escravo_kapa)
    se agencia.tipo == LACO
         entao host = h_exec( agencia.nome, escravo_laco)
    se agencia.tipo == NILO
         entao host = h_exec( agencia.nome, escravo_nilo)
fim
```

Figura 5.6: Algoritmo para ativação dos processos escravos.

Após a criação, os processos escravos aguardam, através da primitiva **h_receive** (recepção bloqueante) que o mestre repasse as informações sobre as agências que devem simular, além dos dados para inicialização. O mestre mantém internamente uma estrutura de dados idêntica à da versão centralizada, para armazenar tanto as informações das agências, quanto as informações das entradas primárias e estímulos.

A Rede de Agências Primitivas é a estrutura que armazena as informações sobre as agências. Consiste de uma lista encadeada contendo todas as agências primitivas que compõem o modelo de simulação. A figura 5.7 mostra a estrutura de dados que representa a rede de agências. Em cada nodo estão as seguintes informações:

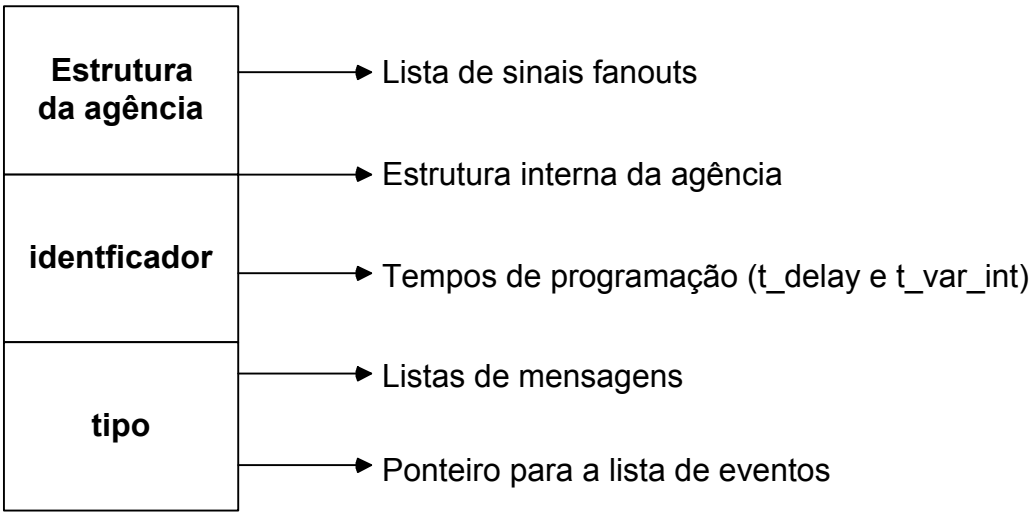

Figura 5.7: Rede de Agências.

Identificador da agência;

Lista com os sinais de interface da agência. Esta lista contém todos os sinais que fazem parte da interface da agência. Para cada sinal de saída ou sinais bidirecionais tem-se, ainda, a lista dos sinais fanouts do sinal. é a partir dessas informações que o mestre poderá propagar as mudanças nos sinais de interface da agência. Além disso, a lista de sinais fanouts deve considerar a possibilidade de um sinal ser composto ou decomposto para formar sinais de largura maior ou menor;

Estrutura interna da agência. A estrutura interna da agência, gerada por um compilador ou editor é particular de cada linguagem e é armazenada na base de dados. O ambiente de simulação busca a estrutura correspondente a cada agência e envia ao mestre como um dos itens da rede de agências;

Os próximos tempos de programação. Nesse item são armazenados dois campos t_delay e t_var_int. O primeiro é o próximo tempo em que a agência está programada na lista de eventos. Já o segundo indica se a agência está ou não programada para o tempo atual por alguma variação de sinal de interface. Ambos os campos tem a função de agilizar a manipulação da lista de eventos com informações auxiliares;

Lista de mensagens. é uma lista contendo as mensagens que devem ser enviadas ao escravo e consiste de uma lista encadeada contendo o código da mensagem e a lista de parâmetros correspondente.

Ponteiro para o nodo onde está programada a agência na lista de eventos. Essa informação é usada quando se deseja retirar a agência do lugar onde está e antecipar a sua programação para um tempo mais recente.

Outras estruturas importantes são a lista de sinais primários e a lista de estímulos. A lista de sinais primários contém a lista de sinais primários do modelo de simulação. A cada sinal está associada uma lista com os respectivos fanouts e o identificador do estímulo que está associado àquele sinal. Já a lista de estímulos é uma série de formas de

onda que serão aplicadas nos sinais primários. Cada estímulo contém um identificador, e informações como a periodicidade e a duração da onda. A figura 5.3, mostrada na seção 5.1, ilustra a estrutura que modela a lista de estímulos.

5.2.3 Lista de eventos

Internamente, o mestre continua mantendo uma lista de eventos ordenada por tempo, indicando os próximos eventos programados para cada agência. Ao consultar esta lista e descobrir algum evento para uma agência, o mestre manda uma mensagem para o processo que a está simulando, informando o evento ocorrido.

A lista de eventos consiste de uma estrutura de dados que armazena os eventos que devem ocorrer ordenados por tempo. Existe, portanto, uma lista para cada tempo onde há um evento a ocorrer. Essas listas estão ligadas a uma roda de tempo circular, como mostra a figura 5.8.

O tempo de simulação vai avançando unitariamente, apontando para os nodos da roda de tempo. Quando o valor do tempo de simulação for maior que o tamanho da roda de tempo, deve-se usar a seguinte expressão para calcular o índice na roda onde está a lista de eventos daquele tempo:

```
índice = tempo_desejado mod tamanho_da_roda
```

A figura 5.9 mostra onde encontrar um evento programado para o tempo 234 em uma roda de tempo de 100 nodos.

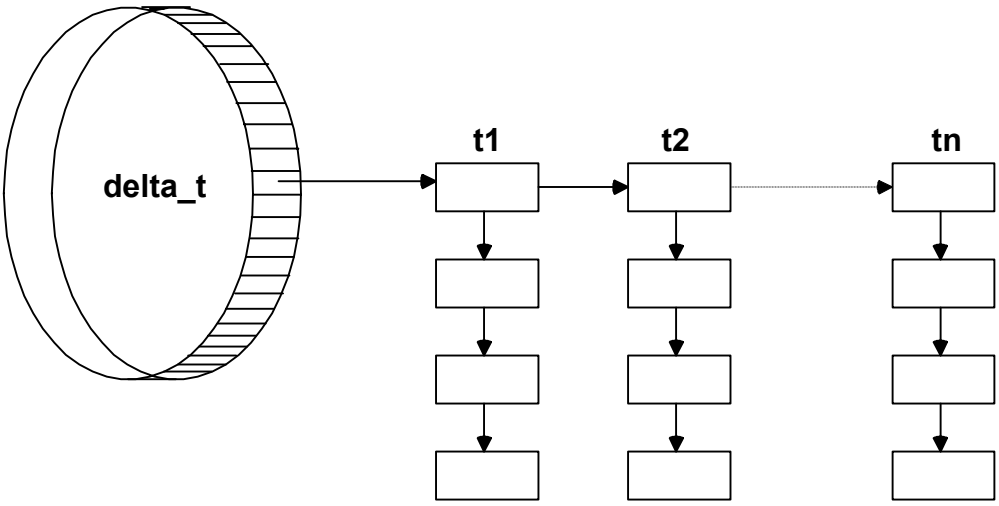

Figura 5.8: Roda de tempo.

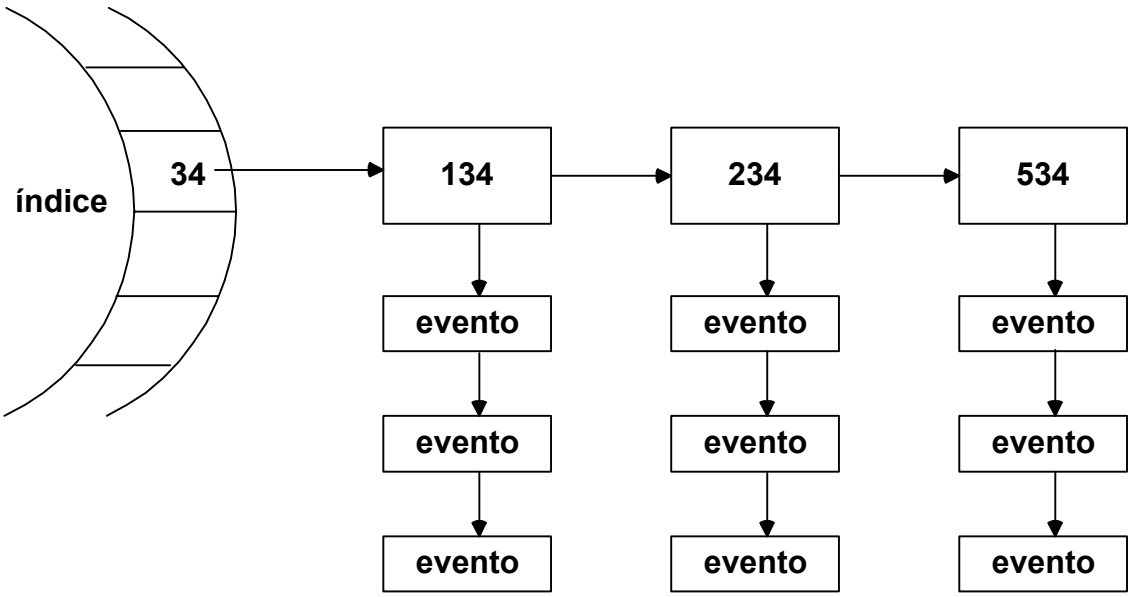

Figura 5.8: Eventos na roda de tempo.

5.2.4 Algoritmo de simulação do mestre

O algoritmo de simulação do mestre continua praticamente igual à versão centralizada, acrescentando apenas alguns ajustes para tratamento dos processos remotos. Depois de receber do ambiente o modelo de simulação e as informações para inicializar o modelo, o mestre cria os processos para simulação e envia as informações iniciais, usando a primitiva **h_send**.

Após, há uma fase de interação com o ambiente para troca de informações sobre sinais. As mensagens trocadas e as primitivas usadas são detalhadas adiante. Uma das mensagens recebidas pelo mestre indica o início efetivo da simulação. A partir daí, tem início a parte principal do algoritmo de simulação.

O início do algoritmo de simulação se dá com o mestre programando, na Lista de Eventos, os eventos provocados por estímulos nas entradas primárias. O mestre, então, avança o tempo até o primeiro evento a ser simulado e, como no início somente existem eventos correspondentes a estímulos externos, programa, para o tempo atual, as agências que são afetadas pela variação do sinal especificado, enviando a elas uma mensagem de variação de sinal de interface.

Após a simulação de cada agência, o mestre deve verificar que mensagens foram retornadas dos escravos e propagar as mudanças nos

sinais de interface entre as agências. Na versão centralizada, era simples executar tal tarefas, uma vez que a chamada de um escravo consistia na chamada de uma função que, no seu retorno, trazia as mensagens de volta. O mestre ficava, então, "bloqueado", esperando a execução da função.

Na versão semi-distribuída, o mestre não pode ficar esperando apenas por um escravo, pois, ao mesmo tempo, existem vários processos escravos sendo executados e que podem retornar resultados. Para conseguir atender a todos, o mestre deve implementar uma fila de mensagens para receber as mensagens retornadas dos processos ativos. Cada mensagem que chega traz o nome do processo que a enviou. Com isso, o mestre pode identificar o remetente e proceder o tratamento das mensagens de retorno.

Depois de receber o retorno de todos os processos escravos para um determinado tempo, o mestre verifica se foi satisfeita alguma condição de parada. Como já foi mostrado na seção 3.1, a simulação pode parar por três motivos: término do tempo de simulação, condição de *breakpoint* satisfeita ou erro grave. Em quaisquer dessas situações, o mestre retorna ao ambiente o tempo de simulação e as mensagens acumuladas até aquele momento. O mestre não encerra o processamento dos escravos, pois a simulação pode continuar. Há uma mensagem específica enviada pelo ambiente que determina o encerramento total da simulação e que faz com que o mestre termine todos os processos escravos ativos.

Se não houver motivo para interromper a simulação, o mestre incrementa o tempo e passa para os próximos eventos da lista. Os passos mencionados acima são repetidos até que surja uma condição de parada. O tratamento dos eventos continua idêntico ao da versão centralizada, ou seja, como foi mostrado na seção 3.1.

5.2.5 Os escravos

O algoritmo dos escravos, abstraindo-se o tratamento específico de cada nível de abstração, consiste na recepção de uma mensagem enviada pelo mestre, processamento e envio de mensagens de retorno para a fila de espera do mestre.

5.2.6 Término

O término da simulação poderá ser provocado pela avaliação verdadeira de uma condição de *breakpoint*, pelo término do tempo de simulação ou pela chegada de uma mensagem de erro fatal de algum escravo. Em qualquer caso, o mestre apenas retorna uma mensagem para o ambiente, informando o tempo atual, a condição de parada e aguardando novas mensagens, não desativando os escravos. Isto se explica pela possibilidade da continuação da simulação após a interação com o usuário. Para o término final da sessão de simulação, é necessária uma mensagem explícita do ambiente. Para isso, deverá ser acrescentada

uma nova mensagem à lista de mensagens possíveis entre mestre e ambiente, para que o mestre desative os escravos e encerre o seu processamento.

5.2.7 Comunicação

A comunicação entre mestre e ambiente de simulação se dá pela troca de mensagens através das primitivas de HetNOS. Depois de ativado, o mestre aguarda, com a primitiva **h_receive**, uma mensagem do ambiente contendo o modelo a ser simulado e as informações para inicialização das agências. Após a chegada dessa mensagem e do seu processamento, o mestre volta a aguardar novas mensagens do ambiente, entre elas aquela que irá provocar o início da simulação. Nesse momento, o ambiente fica bloqueado, aguardando, então, o final da simulação e as mensagens de retorno.

Entre mestre e escravos, a troca de mensagens é similar. O mestre ativa os processos escravos, que ficam bloqueados aguardando as mensagens do mestre (**h_receive**). A cada mensagem, os escravos tratam os eventos relacionados e, após, enviam uma mensagem ao mestre com as informações de retorno (**h_nonblk_send**).

O mestre, por sua vez, envia suas mensagens aos escravos com a primitiva **h_nonblk_send** e as recebe através de uma fila de mensagens de entrada. Duas primitivas são usadas para tratamento dessa fila: **h_nonblk_count_messages**, que verifica quantas mensagens estão

disponíveis na fila, e **h_nonblk_receive**, que recebe uma mensagem disponível na fila.

As mensagens enviadas de ambiente para mestre continuam as mesmas da versão centralizada (**inject, force, put, trace, short, break, read, save** e **simula**), com o acréscimo de uma mensagem chamada **encerra**, que faz com que o mestre conclua o processamento dos escravos, permitindo, assim, o término da sessão de simulação. As mensagens enviadas de mestre para ambiente continuam inalteradas (**trace, break, erro**).

As mensagens trocadas entre mestre e escravos continuam as mesmas. De mestre para escravos: **inicialize_agência, varie_sinal_interface, mude_sinal, force_sinal, varie_clock, leia_sinal, monitora_sinal, salva_estado**. De escravos para mestre: **variei_sinal_interface, variei_clock, programei_evento, devolvo_monitorados, erro**.

5.2.8 Diferenças básicas entre a versão centralizada e a versão semi-distribuída

A diferença entre o algoritmo centralizado e o algoritmo semi-distribuído está nos seguintes aspectos:

- Na versão centralizada, havia apenas uma instância de cada simulador escravo e o mestre efetuava a troca de agências para aquele simulador, quando ele estivesse disponível. Na versão semi-distribuída, cada agência primitiva é simulada por uma cópia do simulador correspondente. Isto significa que cada agência primitiva pertencente à rede de agências é simulada por um processo independente e há tantos processos quantos necessários para a simulação paralela de todas as agências primitivas.

- Na versão semi-distribuída, ao contrário da versão centralizada, o mestre tem que estar munido de recursos que lhe permitam criar e terminar os processos escravos, já que eles estarão sendo executados em máquinas distribuídas pela rede. Na versão centralizada, todos os processos executavam na mesma máquina.

- Na versão semi-distribuída, além do mestre, os simuladores escravos também terão que ter recursos para a troca de mensagens, uma vez que não mais poderão compartilhar memória (pois estarão em máquinas separadas, com memórias distintas).

- Na versão semi-distribuída, uma mensagem específica foi criada para encerramento da sessão de simulação. Durante toda

a sessão, o mestre e escravos ficam ativos nos seus processadores.

5.3 Análise comparativa das versões distribuídas

A análise comparativa das versões propostas deve confrontá-las em dois aspectos:

- complexidade dos ajustes a serem feitos no mestre e nos escravos, ou seja, esforço de desenvolvimento de uma versão distribuída, e
- eficiência da simulação, ou seja, aumento da velocidade esperado em relação à versão centralizada.

5.3.1 Distribuição total conservativa X distribuição total otimista

Em termos de distribuição dos processos, as duas versões se equivalem, já que ambas prevêem a independência dos processos de simulação, sem nenhum controle centralizado. Analisando apenas este aspecto, poder-se-ia considerar que há uma grande exploração do potencial de paralelização da simulação. No entanto, os mecanismos de sincronização utilizados nas duas versões comprometem o seu desempenho.

A versão totalmente distribuída otimista, com seus mecanismos de retrocesso, exige o dispêndio de tempo de processamento para tratamento dos conflitos de tempo. Além do tempo, há a necessidade de gastos de memória com estruturas de dados adicionais para armazenamento dos estados intermediários das agências para fins de retrocesso.

Além disso, a implementação da abordagem otimista requer grandes modificações nos códigos já existentes, para torná-los aptos a salvar periodicamente seus estados, além de mecanismos para retroceder as atividades e enviar antimensagens.

Já a abordagem conservativa, embora de mais fácil implementação, limita o avanço do tempo de simulação pelo processo mais lento, ou seja, o processo cuja execução esteja mais atrasada determina o andamento dos demais processos. Portanto, a possível vantagem obtida com a paralelização dos processos escravos pode ficar comprometida, caso a distribuição da carga entre os processos seja muito discrepante.

Além disso, o fluxo de mensagens nulas na rede, para evitar *deadlocks*, pode ser intenso a ponto de prejudicar o desempenho do sistema.

Outro problema, que atinge tanto a abordagem otimista quanto a conservativa, é a necessidade de consenso de barramento e de

adaptação de sinais. A necessidade desses recursos, característica da simulação multinível, acarreta a utilização de mecanismos complexos, principalmente em se tratando de uma implementação distribuída. Para evitar qualquer controle global para tratar esses problemas, é necessária uma comunicação intensa entre os processos. Por outro lado, qualquer elemento centralizador que seja inserido no sistema para facilitar a solução desses problemas acarretará um desvio da abordagem totalmente distribuída.

5.3.2 Distribuição total conservativa X distribuição parcial

As duas versões em questão assemelham-se pela forma com que implementam o sincronismo entre os processos. A versão totalmente distribuída conservativa sincroniza os processo bloqueando-os à espera de uma condição segura para tratar os eventos, isto é, limitando o processamento dos eventos pelo menor tempo entre eles. A versão parcialmente distribuída, embora de forma diferente, também limita o andamento do tempo de simulação pelo processo mais demorado. Assim, em ambas as versões, os processos são sincronizados por bloqueios.

A diferença entre as versões, que está na existência ou não de um elemento centralizador, implicando uma menor ou maior paralelização, torna-se menos importante quando analisadas as características da simulação multinível, como a necessidade de

mecanismos de consenso de barramento e de adaptadores de sinais entre os diversos níveis de descrição.

Como já foi mostrado anteriormente, a implementação distribuída de mecanismos que resolvam tais problemas é bastante complexa, acarretando um processamento extra que pode prejudicar o desempenho do sistema. O fato da versão parcialmente distribuída manter um elemento centralizador (o mestre) que executa tais tarefas pode ser bem mais uma vantagem do que uma desvantagem. Mesmo a intermediação que o mestre exerce entre os processos, recebendo e enviando mensagens entre os processos, não deve causar tanto impacto na atividade dos escravos quanto a necessidade de tráfego de mensagens nulas para detecção de *deadlocks*.

5.4 Análise qualitativa da influência da granularidade dos módulos

É difícil determinar, *a priori*, na simulação multinível, a carga que cada processo estará processando, pois o sistema envolve módulos de níveis de abstração diferentes e, portanto, de complexidade diferentes. No entanto, é possível imaginar o comportamento de um sistema digital, descrito em uma determinada linguagem de descrição, e dividido em módulos, sendo simulado pelas três versões acima descritas.

A análise feita aqui baseia-se no grau de atividade interna dos módulos do sistema digital, quando a granularidade aumenta ou diminui. As suposições feitas são puramente intuitivas, uma vez que não existem dados concretos sobre os quais pudessem ser feitos estudos mais aprofundados.

Apesar das suposições feitas serem coerentes e provavelmente corresponderem à maioria dos sistemas, efeitos contraditórios poderão ocorrer, dependendo do tipo de circuito testado e de sua atividade interna. Não é possível, portanto, prever os casos excepcionais que poderão ocorrer caso a distribuição de atividade no circuito não seja uniforme.

Supõe-se que um sistema digital tenha sido descrito em um determinado nível de abstração e, após, tenha sido dividido em módulos. Se a divisão foi feita mantendo-se módulos grandes, diz-se que a granularidade do sistema é alta. No entanto, se a divisão foi feita em módulos pequenos, diz-se que a granularidade é baixa.

Se a granularidade é baixa, conseqüentemente a quantidade de módulos necessária é maior, implicando em menos atividade interna para cada módulo e a necessidade de maior troca de informações entre os módulos.

A simulação do sistema descrito dessa forma na versão totalmente distribuída otimista seria menos sujeita a conflitos de tempo,

já que com módulos menores, haveria uma distribuição mais equilibrada da atividade global entre os módulos. Assim, os módulos manteriam seus processamentos em velocidades semelhantes, evitando discrepâncias de tempos. Porém, se houvesse algum conflito de tempo entre os módulos, muitas mensagens deveriam ser desfeitas, acarretando um grande desperdício de tempo.

Já a simulação do sistema na versão totalmente distribuída conservativa faz prever poucos períodos de bloqueio, considerando-se que a quantidade de mensagens entre os módulos é grande e que um módulo ficaria pouco tempo bloqueado, esperando mensagens. Outro fator que faz prever o pouco tempo de bloqueio é o equilíbrio na quantidade de atividade dos módulos.

Com a versão semi-distribuída, a simulação teria o mestre centralizando um fluxo grande de mensagens, que estaria circulando entre os módulos, além de ter de gerenciar muitos processos ativos a cada tempo de simulação.

Por outro lado, se a granularidade é alta, então são necessários menos módulos para compor o sistema. No entanto, cada um desses módulos contém um nível de atividade grande e a troca de mensagens entre os módulos é baixa.

A simulação do sistema descrito com granularidade alta na versão totalmente distribuída otimista tem uma maior possibilidade de

conflito de tempos. Já que o sistema está dividido em módulos grandes, a distribuição da atividade entre os módulos possivelmente não está feita de maneira uniforme. Assim, a chance de alguns módulos terem maior atividade que os demais é grande, acarretando uma maior possibilidade de ocorrência de conflito de tempos. Se realmente ocorrer um conflito de tempo que exija um retrocesso nas atividades, o prejuízo em termos de tempo será grande pois, apesar de haver poucas mensagens a desfazer, há muitos eventos a retroceder em cada módulo.

Na versão totalmente distribuída conservativa, haveria uma grande possibilidade dos processos ficarem um longo tempo bloqueados, já que há poucas mensagens circulando na rede e a distribuição da atividade nos módulos não é uniforme. Assim, um processo com menos atividade interna (e, portanto, mais rápido) logo terá que se bloquear à espera das mensagens de algum módulo mais lento.

Por fim, a simulação com a versão semi-distribuída faria com que o mestre tivesse que tratar menos mensagens, ficando o gasto maior de tempo no processamento interno dos módulos.

A análise feita aqui considera que o sistema todo está descrito em uma mesma linguagem, isto é, em um mesmo nível de abstração, sem determinar qual seja este nível. Considerando os níveis de abstração oferecidos em AMPLO (sistema, transferência entre registradores e portas lógicas), a análise tornar-se-ia bem mais complexa se fosse considerado cada um desses níveis, considerando as suas características particulares.

Além disso, a complexidade aumentaria se fosse considerado um sistema descrito em níveis de abstração diferentes.

6 CONSIDERAÇÕES FINAIS

A simulação é a forma mais utilizada para validação de sistemas eletrônicos. Porém, há problemas computacionais inerentes ao uso desse recurso, como a relação que há entre o tempo de processamento e a acuracidade desejada.

A simulação multinível tenta minimizar o tempo de processamento das simulações, mas não consegue a eficiência desejada quando se trata de grandes circuitos.

Uma forma de aumentar a eficiência da simulação multinível é a agregação de recursos para a paralelização do processamento. Assim, surge a simulação multinível distribuída.

A simulação multinível distribuída combina as facilidades da simulação multinível, como a descrição das partes do circuito em níveis de abstração diferentes, com os recursos do processamento distribuído, que permitem dividir o processamento por diversos processadores numa rede.

Com o circuito dividido em diversos processadores, o aspecto mais importante que se apresenta é a sincronização dos eventos. São duas as principais técnicas para sincronização de eventos em sistemas distribuídos e que, por analogia, são usadas na simulação multinível distribuída: a técnica otimista e a técnica conservativa.

A técnica otimista trata os conflitos de tempo entre os eventos com o retrocesso das atividades dos processos e o envio de antimensagens para anular as mensagens enviadas erroneamente.

Já a técnica conservativa não permite que ocorram conflitos entre os tempos dos eventos, pois os processos ficam bloqueados aguardando um momento seguro de seguirem seu processamento.

No CPGCC da UFRGS, o projeto AMPLO faz uso de simulação multinível não distribuída de sistemas digitais. Nessa perspectiva, os simuladores do AMPLO serviram de base para a análise de requisitos para uma versão distribuída, objeto dessa dissertação.

Os recursos do Sistema Operacional de Rede Heterogêneo HetNOS, desenvolvido no CPGCC/UFRGS, são perfeitamente adequados às necessidades de implementação de um sistema distribuído. Por isso, esta foi a plataforma escolhida para a análise.

Três alternativas para o projeto de distribuição dos simuladores do AMPLO foram propostas, considerando-se o nível de distribuição desejado e o tipo de sincronização utilizada.

A versão totalmente distribuída com sincronização otimista é a que, teoricamente, oferece a melhor distribuição dos processos, evitando qualquer controle centralizado. Porém, o tratamento dos conflitos de tempo através do retrocesso das atividades acarreta um grande *overhead*.

A versão totalmente distribuída com sincronização conservativa também proporciona um grande paralelismo entre os processos, embora o método de sincronização usado faça com que os processos fiquem bloqueados aguardando um tempo seguro para prosseguirem. Este bloqueio pode prejudicar o desempenho do sistema, já que a velocidade de execução dos processos é limitada pelo processo mais lento.

O uso das versões totalmente distribuídas também dificulta a solução de dois problemas característicos da simulação multinível: o consenso dos barramentos e a adaptação de sinais entre módulos de diferentes níveis de abstração.

A versão parcialmente distribuída não proporciona um completo paralelismo entre os processos, uma vez que mantém um processo centralizador (chamado mestre) que faz a intermediação das mensagens entre os processos, além de controlar o avanço do tempo de simulação. No entanto, a sua concepção facilita a solução do problema do consenso dos barramentos e da adaptação dos sinais.

O estudo dos recursos de *hardware* e *software* oferecidos pelo CPGCC, bem como das técnicas de distribuição, mostra que uma implementação distribuída dos simuladores do AMPLO deve seguir as etapas abaixo:

1. O primeiro passo seria partir da versão centralizada e transformá-la em uma versão parcialmente distribuída, uma vez que essa transformação exigiria poucas modificações no código dos simuladores já existentes. Além disso, a adaptação progressiva para uma versão distribuída trará experiência à equipe de desenvolvimento;

2. A seguir, seria desenvolvida a versão totalmente distribuída com sincronização conservativa a partir da versão parcialmente distribuída.

3. Finalmente, a versão do item anterior serviria de base para a implementação da versão totalmente distribuída com sincronização otimista, a mais complexa de ser desenvolvida, por seus mecanismos de retrocesso.

A passagem da versão centralizada dos simuladores para a versão parcialmente distribuída, como proposto no item 1, deverá ser feita utilizando-se as primitivas de HetNOS, que oferecem todos os recursos necessários à implementação.

BIBLIOGRAFIA

[ACP89] Universities Research Association - Fermi National Accelerator Laboratory. **ACP Cooperative Processes - User Manual**, 1989.

[AGR91] Agre, J. R & Tinker, P. A. Useful extensions to a Time warp Simulation System. **Proc. SCS Multiconference on Distributed Simulation**, v. 23, n. 1, Jan., 1991, pp. 78-85.

[BAE89] Baezner, Dirk et al. Algorithmic optimization of simulations on Time Warp. **Proc. SCS Multiconference on Distributed Simulation**, v. 21, n. 2, March., 1989, pp. 73-78.

[BAR92] Barcellos, Antônio Marinho Pilla. **O Sistema Operacional de Rede Heterogêneo HetNOS**. Porto Alegre: CPGCC da UFRGS, Abril 1992, 213 p.il. Dissertação (Mestrado).

[BRY79] Bryant, R. E. Simulation on a distributed system. In **Proc. of the First Inter. Conference on Distributed Computing Systems**, IEEE, 1979.

[CHA79] Chandy, K. Mani & Misra, J. Distributed Simulation: A case study in design and verification of distributed programs.

IEEE Transactions on Software Engineering, v. SE-5, n.5, Sept., 1979, pp. 440-452.

[CHA81] Chandy, K. Mani & Misra, J. Asynchronous Distributed Simulation via a Sequence of Parallel Computations. **Communications of the ACM**, v. 24, n.11, April, 1981, pp. 198-206.

[CHA89] Chandy, K. Mani & Shermann, R. Space-Time and the simulation. **Proc. SCS Multiconference on Distributed Simulation**, v. 21, n. 2, March., 1989, pp. 53-57.

[FID91] Fidge, Colin. Logical Time in Distributed Computing Systems. **Computer**, v. 24, n. 8, Aug. 1991, pp. 28-23.

[FIG90] Figueiró, J. P. **Simulador Mestre para o Sistema AMPLO**. Porto Alegre, Instituto de Informática da UFRGS, dezembro, 1990. (Trabalho de Conclusão)

[FIG92] Figueiró, J. P. **Sincronização de eventos em sistemas distribuídos**. Porto Alegre, PGCC da UFRGS, março, 1992. (Trabalho Individual)

[FRE88] Freitas, C. M. D. S. et al. O Sistema AMPLO: um ambiente integrado para projeto de sistemas digitais. In: Seminário

Integrado de Software e Hardware, 15. Rio de Janeiro, 17-22 julho 1988. **Anais**. Rio de Janeiro, SBC, 1988, p.272-284.

[FUJ90] Fujimoto, Richard M. Parallel Discrete-Event Simulation. **Communications of the ACM**, v. 33, n. 10, Oct. 1990, pp. 34-53.

[GAR91] Garcia, Leonardo de Carvalho. **Arquitetura e Software do Multiprocessador ACP**. CPGCC/UFRGS, Porto Alegre, 1991.

[GOL89] Golendziner, L. G. & Wagner, F. R. & Freitas, C. M. D. S. Modeling digital systems in an integrated design environment. J. Darringer e F. Rammig (eds), 9th International Conference on Computer Hardware Description Languages and their Applications. Washington, EUA, 19-21 junho 1989. **Proceedings**, North-Holland, 1989. pp 147-156.

[HON89] Hontalas, Philip et. al. Performance of the Colliding Pucks simulation on the Time Warp Operating Systems (part 1: Asynchronous behavior and sectoring) **Proc. SCS Multiconference on Distributed Simulation**, v. 21, n. 2, March., 1989, pp. 3-7.

[JEF85] Jefferson, David R. Virtual Time. **ACM Transactions on Programming Languages and Systems**, v.7, n.3, July, 1985, pp. 404-425.

[JEF87] Jefferson, David et al. Distributed Simulation and the Time Warp Operating System. **ACM Operating Systems Review**, nov. 1987.

[LAM78] Lamport, Leslie. Time, Clocks and the Ordering of Events in a Distributed System. **Communications of the ACM**, v.21, n.7, July, 1978, pp. 558-565.

[LEF89] Le Faou, C. & Silva Filho, J. F. **Simulador LAÇO para o sistema AMPLO**. Porto Alegre, CPGCC - UFRGS, maio 1989. (RP no. 112)

[LIN90] Lin, Yi-Bing & Lazowska, Edward D. Optimality Considerations of Time Warp Parallel Simulation. **Proc. SCS Multiconference on Distributed Simulation**, v. 22, n. 1, Jan., 1990, pp. 29-34.

[LIU90] Liu, L.Z. Tropper, C. Local deadlock detection in distributed simulations. **Proceedings of the Multiconference on Distributed Simulation**, v. 22, n. 1, Jan. 1990, pp. 137-143.

[MAD89] Madisetti, Vijay et alii. Efficient Distributed Simulation, **22nd Annual Simulation Symposium**, 1989, pp. 5-21.

[MIS86] Misra, Jayadev. Distributed Discrete-Event Simulation. **Computing Surveys**, v. 18, n. 1, March, 1986, pp. 39-65.

[NAJ87] Najjar, Wallid et ali. Parallel Discrete-Event Simulation. **IEEE Design and Test of Computers**, Dec. 1987, pp. 41-44.

[REI90] Reither, Peter et al. Cancellation Strategies in Optimistic Executions Systems. **Proc. SCS Multiconference on Distributed Simulation**, v. 22, n. 1, Jan., 1990, pp. 112-121.

[SIL88] Silva Filho, J. F. & Wagner, F. R. & Le Faou, C. **LAÇO - Uma linguagem para descrição de hardware no nível de sistema**. Porto Alegre, PGCC da UFRGS, outubro 1988. (RP no. 094)

[SMI86] Smith, R. Fundamentals of Parallel Logic Simulation. In: Design Automation Conference, 23. **Proceedings...**, p. 2-11, 1986.

[SU 89] Su, Wen-King & Seitz, Charles. Variants of the Chandy-Misra-Bryant distributed discrete-event simulation algoritm. **Proc. SCS Multiconference on Distributed Simulation**, v. 21, n. 2, March., 1989, pp. 38-43.

[SUN90] SUN Microsystems, Inc. **Network Programming Guide**, 1990.

[WAG87] Wagner, F. R. **KAPA - Uma linguagem para a descrição de hardware no nível de transferência entre registradores**. Porto Alegre, PGCC da UFRGS, abril 1987. (RP no. 068)

[WAG87a] Wagner, F. R. & Freitas, C. M. D. S. **NILO - Uma linguagem de descrição de hardware no nível de portas lógicas**. Porto Alegre, PGCC da UFRGS, março 1987. (RP no. 066)

[WAG87b] Wagner, F. R. & Freitas, C. M. D. S. & Golendziner, L. G. **Linguagens de descrição de hardware para suporte à integração do processo de projeto em AMPLO**. Porto Alegre, PGCC da UFRGS, março 1987. (RP no. 065)

[WAG87c] Wagner, F. R. & Freitas, C. M. D. S. & Golendziner, L. G. A Digital System Design Methodology based on Nets os Agencies. In: M. Barbacci e C. J. Koomen (eds.) **8th IFIP International Conference on computer Hardware Description Languages and their Applications.** Amsterdam, Holanda, 27-29 abril 1987. North-Holland, 1987, pp. 213 e 224.

[WAG88] Wagner, Flávio R. et alii. The AMPLO system: an integrated environment for digital systems design. In: F.RAMMIG (ed), **IFIP Workshop on Tool Integration and Design**

Environments, Paderborn, R. F. Alemanha, North-Holland, 1988. pp 221-232.

[WAG88a] Wagner, F. R. AMPLO - um ambiente integrado para projeto de circuitos VLSI. In: **Congresso da Sociedade Brasileira de Microeletrônica**, III. São Paulo, SP, 12-14 julho 1988. Anais, USP, 1988. pp 437-446.

[WAG88b] Wagner, F. R. O modelo de redes de agências e sua aplicação na simulação de sistemas digitais. In: **Workshop Brasileiro de Simulação**, II. São José dos Campos, SP, 1 - 2 setembro 1988. Anais. INPE, 1988. pp 81-88.

[WAG89] Wagner, F. R. Um ambiente integrado para simulação de sistemas digitais. In: Simpósio Brasileiro de Concepção de Circuitos Integrados, IV. Rio de Janeiro, RJ, 12-14 abril 1989. **Anais**. SBC/UFRJ, Rio de Janeiro, 1989.

[WAG90] Wagner, P. R. **Um ambiente integrado de simulação de sistemas digitais.** Porto Alegre, PGCC da UFRGS, novembro, 1990. (Dissertação de mestrado).

[WIL87] Wilson, Andrew. Parallelization odf an Event Driven Simulator for Computer Systems Simulation. **Simulation**, v. 49, n. 2, 1987, pp. 72-78.