Antonio Ribeiro Alves Júnior

Arquitetura de um *Middleware* de Comunicação entre Processos para Implementação de Simulação Distribuída

Itajubá - MG
Abril de 2014

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

Antonio Ribeiro Alves Júnior

Arquitetura de um *Middleware* de Comunicação entre Processos para Implementação de Simulação Distribuída

Dissertação submetida ao Programa de Pós-Graduação em Ciência e Tecnologia da Computação como parte dos requisitos para obtenção do Título de Mestre em Ciências em Tecnologia da Computação

Área de Concentração: Sistemas de Computação

Orientador:
Prof. Dr. Edmilson Marmo Moreira

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

Abril de 2014
Itajubá/MG

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

Antonio Ribeiro Alves Júnior

Arquitetura de um *Middleware* de Comunicação entre Processos para Implementação de Simulação Distribuída

Dissertação aprovada pela banca examinadora em 30 de abril de 2014, conferindo ao autor o título de **Mestre em Ciências em Tecnologia da Computação**

Banca Examinadora:
Prof. Dr. Edmilson Marmo Moreira (Orientador)
Prof. Dr. Otávio Augusto Salgado Carpinteiro (Co-orientador)
Prof. Dr. Mauro César Bernardes
Prof. Dr. Enzo Seraphim

Itajubá/MG
2014

Este trabalho é dedicado ao meu pai Antonio Ribeiro Alves, exemplo de trabalho, dedicação e bom humor.

Agradecimentos

Acima de tudo agradeço a Deus.

Agradeço ao professor Dr. Edmilson Marmo Moreira que, além de me apoiar e me orientar, me suportou durante esses vinte meses de trabalho, incluindo duas mudanças de cidade.

À Minha esposa Gislene, companheira de todos os momentos e incentivadora maior do meu trabalho.

Aos meus pais, Antonio e Márcia, pelo indubtável apoio em todos esses anos de Unifei.

Aos meus irmãos, que mesmo distante me serviram como fonte de inspiração nesta jornada.

Aos amigos e companheiros de república Maurício Faria, Gustavo Walbon e Tiago Maluta, que foram fontes de inspiração na minha jornada acadêmica. A computação não seria tão interessante e divertida se não fosse a companhia de vocês.

A special thanks for all my colleagues from the *Universiteit Gent*. You make the cold Belgium a bit warmer.

Antonio Ribeiro Alves Júnior

Resumo

Este trabalho apresenta uma proposta de arquitetura na camada de comunicação e mobilidade de processos lógicos para a implementação de simulação distribuída de eventos discretos. Esta camada provê uma comunicação transparente entre os diferentes processos lógicos em um sistema de comunicação, independente da sua localização no sistema distribuído. O mecanismo de migração proposto por esta camada de comunicação baseia-se na serialização de um processo lógico em um determinado ambiente do sistema e o seu envio para o ambiente de destino. Considerando a migração de processos lógicos para a implementação de algoritmos de balanceamento de carga voltados para a simulação distribuída, a camada proposta mantém a comunicação entre os diversos processos lógicos de maneira uniforme e transparente durante e após eventuais migrações para diferentes ambientes sem a necessidade de intervenção para ajuste nos endereços físicos destes processos.

Abstract

This paper presents a proposed architecture of a communication layer and mobility of logical processes for the implementation of distributed discrete events simulation. This layer provides a transparent communication among different logical processes in a communication system, regardless of its location in the distributed system. The migration mechanism proposed by this communication layer is based on the serialization of a logical process in a particular system environment and its delivery to the target environment. Considering logical processes migration to implement load balancing algorithms for distributed simulation, the proposed layer maintains communication between different logical processes in a uniform and transparent way during and after any migration to different environments without the need of intervention adjustments of the physical addresses in these processes.

Sumário

Lista de Figuras

1 Introdução p. 12

1.1 Simulação . . . . p. 13

1.1.1 Simulação Sequencial . . . . p. 13

1.2 Simulação Distribuída . . . . p. 15

1.3 Objetivo . . . . p. 16

1.4 Organização do Documento . . . . p. 17

2 Simulação Distribuída de Eventos Discretos p. 19

2.1 Princípios da Simulação Baseada em Eventos Discretos . . . . . . p. 19

2.2 Categorias de Protocolos de Simulação . . . . p. 21

2.2.1 Protocolos Conservativos . . . . p. 22

2.2.2 Protocolos Otimistas . . . . p. 23

2.3 O Protocolo *Time Warp* . . . . p. 24

2.3.1 Detecção e Tratamento de Inconsistências . . . . p. 25

2.4 Balanceamento de cargas . . . . p. 28

2.4.1 Escalonamento de processos em um sistema distribuído . . p. 29

2.4.2 Migração de processos . . . . p. 30

2.4.3 O Uso de Agentes Móveis Para Prover Mobilidade . . . . . p. 31

2.4.4 Requisitos Para Mobilidade de um Processo Lógico . . . . p. 32

2.5 Utilização do *middleware* de comunicação para o desenvolvimento de um *framework* para simulação distribuída . . . . p. 33

2.5.1 Encapsulamento . . . . p. 34

2.5.2 Transparência . . . . p. 34

2.5.3 Reusabilidade . . . . p. 35

2.6 Soluções Existentes . . . . p. 35

2.7 Considerações finais . . . . p. 36

3 O Projeto do Middleware proposto p. 37

3.1 Arquitetura e separação das camadas . . . . p. 38

3.2 Módulos do *middleware* . . . . p. 38

3.3 A classe *Environment* . . . . p. 42

3.3.1 Estrutura interna . . . . p. 42

3.3.2 Tabela de endereços de processos . . . . p. 43

3.4 A classe *Proxy* . . . . p. 45

3.5 A classe *Process* . . . . p. 48

3.5.1 Ciclo de vida de um processo . . . . p. 49

3.5.2 Serialização de um processo . . . . p. 51

3.5.3 Migração de processos . . . . p. 51

3.5.4 Atualização da tabela de endereços dos processos . . . . . p. 55

3.5.5 Troca de mensagens . . . . p. 55

3.5.6 Comunicação local direta . . . . p. 56

3.5.7 Comunicação indireta . . . . p. 57

3.5.8 Migração e Comunicação Contínua . . . . p. 60

3.6 Considerações Finais . . . . p. 62

4 O Projeto de um Framework de Simulação p. 63

4.1 O Módulo Componente . . . . p. 63

4.2 Componentes Básicos . . . . p. 65

4.2.1 O Componente Fila . . . . p. 66

4.2.2 O Componente Gerador . . . . p. 68

4.2.3 O Componente Consumidor . . . . p. 68

4.3 O *Kernel* do *Framework* . . . . p. 69

4.4 Protocolos de Sincronização . . . . p. 70

4.5 Algoritmos de balanceamento de Carga . . . . p. 70

4.6 Considerações Finais . . . . p. 70

5 Exemplos de Uso p. 72

5.1 Troca de Mensagens . . . . p. 72

5.2 Exemplo Básico de Modelagem . . . . p. 73

5.3 Variação de um Modelo para Comparação . . . . p. 75

5.4 Exemplo de Simulação Distribuída . . . . p. 77

5.5 Considerações Finais . . . . p. 79

6 Discussões Finais e Conclusões p. 80

6.1 Contribuições deste Trabalho . . . . p. 81

6.2 Sugestões para Trabalhos Futuros . . . . p. 81

Referências Bibliográficas p. 83

Lista de Figuras

1.1 Simulação Sequencial . . . . p. 14

1.2 Geração de um novo evento . . . . p. 15

2.1 Inconsistência na simulação . . . . p. 22

2.2 Processo p_2 envia uma mensagem com *timestamp* $t = 5$ para o processo p_1 . . . . p. 25

2.3 Mensagem *straggler* . . . . p. 26

2.4 Envio de anti-mensagem para cancelamento de mensagem . . . . . p. 28

3.1 Camadas da arquitetura do *framework*. . . . . p. 39

3.2 Diagrama de classes. . . . . p. 40

3.3 A arquitetura interna de um *environment*. . . . . p. 43

3.4 Ciclo de vida de um processo lógico. . . . . p. 50

3.5 Diagrama de eventos durante uma migração no ambiente de origem p. 54

3.6 Diagrama de eventos durante uma migração no ambiente de destino p. 54

3.7 Diagrama de eventos durante uma comunicação local direta. . . . p. 57

3.8 Comunicação indireta *proxy-process*. . . . . p. 58

3.9 Diagrama de eventos nos ambiente de origem e de destino durante uma uma comunicação indireta. p. 59

3.10 Diagrama de repasse de mensagens. . . . . p. 61

4.1 A camada aqui denominada *framework*. . . . . p. 64

4.2 Hierarquia dos componentes básicos . . . . p. 66

4.3 Modelagem de uma via urbana. . . . . p. 67

5.1 Modelo com um consumidor . . . . p. 77

5.2 Modelo com dois consumidores . . . . p. 77

1 Introdução

Simulação é a imitação, ao longo do tempo, da execução de um processo real ou de um sistema (BANKS et al., 2010). Uma simulação baseia-se na abstração das características chaves do que se deseja simular, construindo assim um modelo que representa, da maneira mais fiel possível, o comportamento real deste processo ou sistema. A simulação pode ser usada para prever um comportamento quando o sistema real não pode ser comprometido ou está inacessível, ou por ser considerado perigoso ou inaceitável comprometer este sistema. Também pode ser empregada quando se deseja antever o comportamento de um sistema que ainda não foi desenvolvido ou simplesmente não exista (SOKOLOWSKI; BANKS, 2008).

A utilização de computadores para avaliar um modelo representativo de um sistema real leva à chamada simulação computacional. A simulação computacional tem sido amplamente empregada em diversos ramos, como simulação de circuitos elétricos (NAGEL; ROHRER, 1971), simulação eletromagnética (OSKOOI et al., 2010), simulação metereológica (LYNCH, 2007), simulação de serviços (MATLOFF, 2011), simulação de tráfego e trânsito (BHAM; BENEKOHAL, 2004), entre outros.

Em alguns casos, como na simulação do comportamento de circuitos eletromagnéticos e na simulação metereológica, o modelo é baseado em uma simulação numérica (HIGHAM, 1971). Já em casos como a simulação de tráfego e da simulação de serviços, o modelo é baseado em eventos discretos, onde a operação do sistema é representada como uma sequência cronológica de eventos. Este trabalho

aborda a simulação baseada em eventos discretos.

1.1 Simulação

A simulação é uma técnica que permite prever e visualizar o comportamento de sistemas reais a partir de modelos matemáticos. As aplicações da simulação abrangem diversos benefícios, tais como: a possibilidade de antever possíveis problemas ou comportamentos indesejáveis de um sistema, auxiliar na tomada de decisão sem a necessidade de intervir no sistema real, facilitar a manipulação e alteração dos modelos, economizar recursos (físicos e financeiros) durante a tomada de decisões, dentre outros.

Para utilizar a simulação é necessário construir e analisar modelos que representem o sistema. Os modelos podem ser classificados de diferentes formas. Uma classificação pode ser considerada verificando a influência ou não de variáveis aleatórias no sistema. Os modelos que sofrem influência de variações aleatórias são denominados modelos estocásticos, enquanto os modelos que são livres de tal comportamento são denominados modelos determinísticos.

Os modelos que descrevem o comportamento através do tempo podem também ser classificados como contínuos ou discretos no tempo. Nos modelos de estados contínuos, as variáveis de estados variam espontaneamente. Já nos modelos de estados discretos, as mudanças ocorrem em pontos específicos e descontínuos do tempo.

Este trabalho enfoca os modelos estocásticos e de estados discretos, uma vez que eles são os que melhor representam modelos de sistemas computacionais.

1.1.1 Simulação Sequencial

Um sistema de simulação sequencial, onde uma única máquina executa toda a simulação, pode ser retratado como uma fila de eventos aguardando para serem

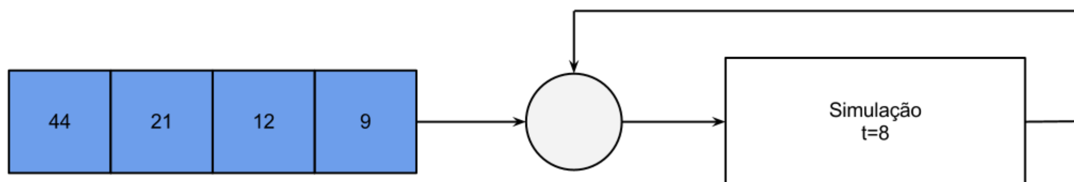

Figura 1.1: Simulação Sequencial

tratados. Cada evento possui o seu tempo de execução, como pode ser visto na figura 1.1, que deve ser obedecido para garantir consistência do resultado.

Neste modelo sequencial, o sistema responsável pela simulação retira o próximo evento da fila de execução para tratá-lo. Ao fim do seu processamento, um próximo evento é retirado da fila, e isto se repete até o final de lista de eventos futuros. O tratamento de um evento pode ou não resultar em dados que influenciem num processamento futuro.

A simulação possui um relógio lógico interno que simboliza o tempo lógico da simulação. A cada avanço deste tempo discreto, o valor do seu relógio lógico é incrementado e é verificado na lista de eventos futuros se existe um evento cujo tempo previsto para a execução é igual ao valor atual do relógio lógico. Caso haja um evento a ser executado naquele determinado momento, este evento é retirado da fila de eventos futuros e é processado pela simulação.

A atualização do relógio lógico sofre influência do evento a ser processado, ou seja, dependendo do modelo simulado, um evento pode demorar mais ou menos tempo para ser processado. Esta variação é levada em conta quando se atualiza o relógio lógico da simulação ao final do processamento de cada evento.

Ainda dependendo do modelo a ser simulado, a execução de um evento pode resultar na criação de um novo evento que deve ser reinserido no sistema. Conforme ilustrado na figura 1.2, ao se executar um evento no tempo $t = 8$, um novo evento foi criado para ser processado no tempo $t = 33$. Quando isso ocorre, o evento é inserido na fila de eventos futuros e aguarda para ser processado no tempo determinado.

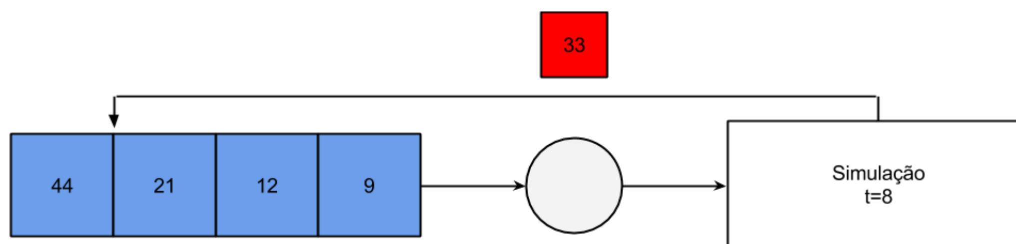

Figura 1.2: Geração de um novo evento

Vale salientar que um processo nunca gera eventos para serem tratados em um tempo lógico inferior ao atual valor do seu relógio lógico interno. Ou seja, se o processo possui um tempo lógico $t = a$, um possível evento gerado em decorrência da execução de um evento neste determinado momento deverá ter um tempo $t = b$ de maneira que $b > a$.

1.2 Simulação Distribuída

A simulação é um processo que apresenta um alto custo computacional, devido a grande quantidade de dados que devem ser processados e a complexidade dos modelos matemáticos empregados. Esses fatores em conjunto podem encarecer computacionalmente o sistema, levando à ineficiência da simulação.

Uma das formas encontrada para solucionar estes problemas foi dividir o tratamento dos diversos eventos entre vários processadores de uma mesma máquina paralela ou em um sistema distribuído, dando origem assim à Simulação Distribuída.

Distribuindo os eventos, reduz-se o tempo gasto pelos programas de simulação, mas, em contrapartida, novas situações necessitam ser observadas devido às características deste tipo de aplicação. É preciso sanar os problemas como a sincronização dos processos distribuídos, sobrecarga da rede de comunicação, necessidade de balanceamento de carga do sistema, dentre outros.

1.3 Objetivo

Conforme descrito, um dos problemas de se distribuir a simulação entre diversos nós de um sistema é a possibilidade de que os processos lógicos não sejam homogeneamente distribuídos, causando um desbalanceamento de carga no sistema. Isto pode ocorrer quando, por exemplo, processos que demandam um processamento mais intenso são agrupados em um mesmo nó, enquanto processos que não demandam tanto processamento são distribuídos pelo sistema. A concentração de processos com alta demanda de processamento em um mesmo nó do sistema levaria à sobrecarga deste nó, causando um desbalanceamento do sistema.

Outra situação que levaria a um desbalanceamento de carga é a existência de processos que se comunicam com uma frequência muito grande executando em nós distintos do sistema, sobrecarregando a rede com troca de mensagens. Se fosse possível detectar os processos que se comunicam com maior frequência e agrupá-los em um mesmo nó do sistema, diminuiria-se consideravelmente o tráfego de mensagens na rede, aumentando o desempenho da simulação.

Para que seja feita a partição dos processos lógicos por entre os nós do sistema, existem algoritmos capazes de analisar o comportamento do sistema, considerando fatores como volume de troca de mensagens e necessidade de processamento de cada processo lógico, para em seguida mapear cada processo para um determinado nó do sistema.

Porém, para que isso seja possível, o processo lógico deve ser capaz de efetuar uma migração, ou seja, deve ser capaz de interromper sua execução em um determinado instante, migrar para um nó diferente do sistema e, de maneira transparente, retornar à execução no ponto em que ela foi interrompida.

Neste contexto, o objetivo deste trabalho é a criação de um *middleware* de comunicação que possibilite a um processo lógico efetuar de maneira transparente a migração de um ambiente de simulação para outro. Este *middleware* deve prover diversas facilidades além da mobilidade do processo lógico, como troca de mensa-

gens entre os processos lógicos, redirecionamento de mensagens transientes (caso o processo tenha migrado no mesmo momento em que recebia uma mensagem), localização de um processo através de um endereço lógico que seja único em toda a simulação (mesmo o processo migrando para outro nó do sistema, deve ser possível comunicar com este processo usando apenas o seu endereço lógico) e serialização do processo lógico, necessário para se salvar o estado de um processo em um determinado momento.

É sugerido por Tavakoli, Mousavi e Komashie (2008) uma plataforma de simulação para desenvolvimento de simulação discreta de eventos. É baseado na mesma idéia de um *framework* genérico que o *framework* aqui proposto é modelado.

Ao contrário do que propõe Park e Fishwick (2010), onde é estipulado uma arquitetura de *hardware* específica, este trabalho tem como arquitetura alvo um *cluster* formado por uma rede heterogênea de computadores, tal como proposto por Cicirelli e Nigro (2013).

Para validar o funcionamento do *middleware* proposto foi desenvolvido também um micro-*framework* utilizando este *middleware* como base para todo o seu desenvolvimento.

1.4 Organização do Documento

O capítulo seguinte traz uma abordagem introdutória à simulação distribuída de eventos discretos e descreve o funcionamento de um protocolo utilizado para a sincronização de processos de um programa de simulação distribuída. O capítulo traz também uma discussão sobre o funcionamento de um *framework* de simulação e trata de assuntos como balanceamento de carga e envio de mensagens.

O capítulo três apresenta a modelagem do *middleware* de comunicação proposto por este trabalho.

O quarto capítulo apresenta a modelagem do *framework* de simulação, exem-

plificando seu uso em conjunto com o *middleware* de comunicação.

Por fim os capítulos cinco e seis trazem, respectivamente, alguns exemplos de utilização do *middleware* e do *framework* aqui propostos e uma discussão final sobre o projeto.

2 Simulação Distribuída de Eventos Discretos

Por demandar um intenso processamento computacional, a simulação por vezes pode se tornar uma opção demasiadamente custosa. Melhores algoritmos e computadores mais modernos são maneiras de tornar a simulação mais eficiente. Em paralelo a isto existe a possibilidade de se distribuir a simulação entre vários nós de um sistema, ou entre vários núcleos de uma máquina paralela. O uso desta técnica permite que certos processos sejam executados simultaneamente, acelerando assim a simulação.

Em contrapartida, ao se dividir uma simulação entre vários nós em um sistema distribuído, são criados diversos novos requisitos ao sistema. Itens como sincronização dos processos, comunicação entre os processos, balanceamento de cargas no sistema, dentre outros, devem ser gerenciados pela aplicação de simulação a fim de se garantir um resultado consistente do processo de simulação.

2.1 Princípios da Simulação Baseada em Eventos Discretos

No paradigma da simulação discreta, três estruturas representam a simulação orientada a eventos:

- As variáveis internas do sistema;

- Uma lista de eventos, denominada lista de eventos futuros. Esta lista abriga os eventos a serem executados;
- Um relógio lógico global, que controla o processo de execução da simulação.

O relógio lógico global armazena o chamado Tempo Virtual Global (*Global Virtual Time* - GVT). O GVT representa o momento atual da simulação. O seu valor sofre incrementos discretos, e cada unidade do GVT representa uma fração real de tempo, discretizada para o modelo a ser simulado.

Cada evento a ser executado possui uma marca de tempo (*timestamp*) que determina quando, no tempo virtual do sistema, este evento deve ser executado. O programa de simulação repetidamente retira o evento com o menor *timestamp* da fila de eventos futuros para processá-lo.

Para a implementação da simulação distribuída, a estrutura da simulação tradicional, inerentemente sequencial, teve que sofrer adaptações para incorporar os conceitos de computação distribuída (REYNOLDS et al., 1992). Para isto, o sistema passou a ser dividido em processos lógicos que representam um processo do sistema real. Portanto uma simulação é um conjunto com n processos lógicos $p_1, p_2, p_3, ..., p_n$, executando em processadores distintos.

Assim como o sistema de simulação centralizado possui um relógio que representa o tempo global da simulação, ao se dividir a simulação em um sistema distribuído, cada processo lógico possui um relógio lógico interno que representa o avanço do tempo da simulação para aquele processo. O relógio de cada processo lógico armazena o que é chamado de Tempo Virtual Local (*Local Virtual Time* - LVT), e é este LVT que indica o tempo virtual de cada processo lógico.

Isso garante que os eventos a serem executados por um determinado processo lógico sejam executados no tempo determinado pelo seu *timestamp*. Portanto, um processo lógico, se olhado isoladamente, possui as mesmas variáveis que um sistema de simulação sequencial: uma fila de eventos a serem executados, um relógio lógico interno e as suas variáveis internas.

Porém, em certos momentos, um processo pode necessitar se comunicar com outro processo lógico. Um processo, por exemplo, pode enviar um evento para ser executado por outro processo lógico. Esta comunicação é feita através de troca de mensagens. Um processo lógico deve ser capaz de enviar mensagens a outros processos lógicos que, por sua vez, devem ser capazes de receber estas mensagens.

Assim como todo evento discreto, um evento enviado de um processo lógico a outro possui um *timestamp* que indica quando este deve ser executado pelo processo lógico em questão. Porém, como os processos lógicos em um sistema distribuído não compartilham um mesmo relógio lógico (cada processo lógico possui seu próprio relógio interno), o sistema de simulação distribuída deve estar preparado para tratar situação de inconsistência da simulação.

Como cada processo lógico possui um relógio lógico interno independente, pode acontecer de um processo estar em um tempo virtual mais atrasado que os demais e enviar um evento para ser processado em um tempo que já foi superado pelo relógio lógico do processo que recebe a mensagem. Assim como ilustrado na figura 2.1, o processo em questão possui seu relógio lógico local no tempo $t = 24$ e recebe uma mensagem para ser processada no tempo $t = 11$. Como o evento a ser processado possui um *timestamp* inferior ao valor do relógio interno do processo lógico, o sistema encontra-se em um momento de inconsistência. Um sistema de simulação distribuída deve ser capaz de contornar (ou evitar) tal situação.

Uma mensagem que contém um evento para ser processado em um tempo pertencente ao passado lógico do processo que a recebeu é denominada Mensagem *Straggler*.

2.2 Categorias de Protocolos de Simulação

Para garantir a sincronização dos processos lógicos na simulação distribuída são usados protocolos de sincronização, que garantem a consistência do sistema ao final da simulação. Os protocolos podem ser divididos em duas categorias quanto

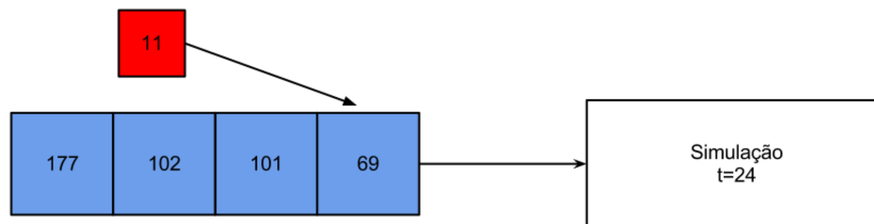

Figura 2.1: Inconsistência na simulação

ao seu comportamento perante uma inconsistência de causa e efeito no sistema: protocolos conservativos e protocolos otimistas.

Um protocolo conservativo evita que um erro de causa e efeito aconteça, garantindo que toda mensagem recebida por um processo lógico esteja dentro do tempo de execução daquele processo. Já um protocolo do tipo otimista não previne a ocorrência de inconsistências no sistema, mas provê mecanismos capazes de recuperar o sistema quando elas ocorrem.

2.2.1 Protocolos Conservativos

Os protocolos conservativos evitam a possibilidade da ocorrência de erros de causa e efeito, ou seja, sua preocupação está em determinar quando é seguro processar um evento. Nas simulações distribuídas sincronizadas com protocolos conservativos, um processo lógico só tratará um evento se puder garantir que não chegará um outro com evento cujo *timestamp* é inferior ao do evento a ser tratado (SRINIVASAN; REYNOLDS JR., 1995).

Os primeiros protocolos de simulação distribuída foram baseados em abordagens conservativas. Esses protocolos, desenvolvidos independentemente por Chandy e Misra (1979) e Bryant(1977) (também tratados neste texto por CMB), exigiam que a especificação dos canais de comunicação entre os processos fosse feita estaticamente e também que as mensagens chegassem em cada canal obedecendo uma ordem crescente dos seus *timestamps*.

Como os canais de comunicação são conhecidos, uma vez que foram previamente especificados antes do início da simulação, é possível, para cada processo, determinar o tempo virtual dos seus processos vizinhos, já que cada mensagem recebida está rotulada com o relógio local do processo emissor. Com essa informação, os processos podem tratar os eventos que possuam tempo de ocorrência menor que o tempo virtual dos canais de comunicação que chegam ao processo.

Um problema pode surgir quando não há mensagem trafegando através de um determinado canal de comunicação. Quando isto ocorre, não há como um processo obter o *LVT* daquele canal, o que pode levar à um *deadlock*. Para contornar tal efeito, é necessário lançar mão de mecanismos, como o mecanismo das mensagens nulas proposto por Misra e Chandy (1979), que força a atualização do LVT dos canais que estão vazios.

Segundo Fujimoto (FUJIMOTO, 1999), a maior desvantagem dos mecanismos conservativos consiste no fato de que estes não podem explorar totalmente o paralelismo disponível na aplicação, uma vez que frequentemente os processos lógicos precisam interromper sua execução a fim de aguardar a chegada de um novo evento, ou mesmo uma mensagem nula.

2.2.2 Protocolos Otimistas

Diferentemente dos protocolos conservativos, os protocolos otimistas não possuem nenhum mecanismo que evite a ocorrência de uma anomalia de causa e efeito. O que acontece em um protocolo otimista é que, uma vez detectado o recebimento de uma mensagem *straggler*, o protocolo deve ser capaz de desfazer a simulação até um tempo virtual onde a mensagem recebida não represente uma inconsistência no sistema, e continuar a simulação a partir deste ponto.

Um dos mais conhecidos protocolos otimistas é o *Time Warp*, que possui algumas implementações, tais como *Jade Time Warp* (BAEZNER; LOMOW; UNGER, 1994), o sistema *SPEE-DES* (*Synchronous Parallel Environment for Emula-*

tion and Discrete Event Simulation)(STEINMAN, 1992), o *WARPED* (MARTIN; MCBRAYER; WILSEY, 1996) e o *Georgia Tech Time Warp (GTW)* (DAS, 1994). O *Time Warp* foi originalmente proposto por Jefferson (JEFFERSON, 1985).

Desenvolvido por Moreira (2003), o protocolo *Rollback* Solidário também utiliza a abordagem otimista para sincronizar os processos lógicos em uma simulação distribuída.

2.3 O Protocolo Time Warp

Assim como na simulação sequencial, um processo lógico na simulação distribuída efetua repetidamente a ação de retirar um evento da fila de eventos futuros e tratá-lo.

O tratamento de um evento pode ou não (dependendo do modelo simulado) gerar novos eventos para serem colocados na fila de eventos futuros. Uma vez gerado um novo evento, caso ele seja destinado ao mesmo processo lógico, este é simplesmente inserido na fila de eventos futuros, obedecendo o seu *timestamp*.

Porém, caso a execução de um evento resulte num novo evento a ser tratado por um processo lógico distinto, o processo cuja execução originou o novo evento deve encaminhar este novo evento para o seu processo correspondente. Para isto, os processos lógicos se comunicam através de troca de mensagens. Quando um processo p_2 necessita encaminhar um evento para ser tratado pelo processo p_1, isto é feito através do envio de uma mensagem de p_2 para p_1 (figura 2.2).

Como os processos lógicos executam de forma independente entre si, não existe nenhuma garantia que os processos p_1 e p_2 possuam o mesmo tempo virtual no momento da troca de mensagem. Caso o LVT do processo p_2 seja superior ao valor do LVT do processo p_1 (como ilustrado pela figura 2.2), existe uma garantia de que o *timestamp* da mensagem enviada por p_2 também seja superior ao valor do LVT de p_1, e isso garante a consistência da simulação neste caso. Mas o mesmo

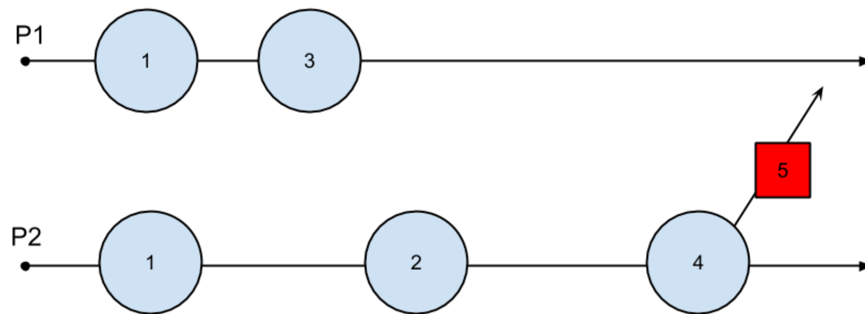

Figura 2.2: Processo p_2 envia uma mensagem com *timestamp* $t = 5$ para o processo p_1

não ocorre quando o LVT de p_2 é inferior ao LVT de p_1. Nesta situação existe a possibilidade de que o *timestamp* da mensagem enviada seja inferior ao LVT do processo p_1, resultando em uma anomalia de causa e efeito.

2.3.1 Detecção e Tratamento de Inconsistências

O protocolo *Time Warp* não possui nenhum mecanismo que evite que uma mensagem com um *timestamp* inferior ao LVT do processo receptor seja entregue. Porém, quando isto ocorre, o processo receptor identifica para qual momento no tempo de simulação (qual LVT) ele deve retornar para que haja uma consistência na simulação. Existe então um intervalo de tempo de simulação que fica entre o LVT em que o processo recebeu a mensagem *straggler* e o tempo virtual referente ao *timestamp* da mensagem recebida. Este intervalo de tempo é denominado Janela de Retorno.

Assim que for identificado o LVT que garante a consistência da simulação para aquele processo lógico, este restaura a simulação para este determinado tempo virtual, realizando um retorno (*rollback*) da sua simulação. Para isso, todos os eventos tratados dentro da janela de retorno devem ser descartados e devem ser novamente tratados pelo processo lógico que efetuou o *rollback*, porém agora após o tratamento da mensagem recebida.

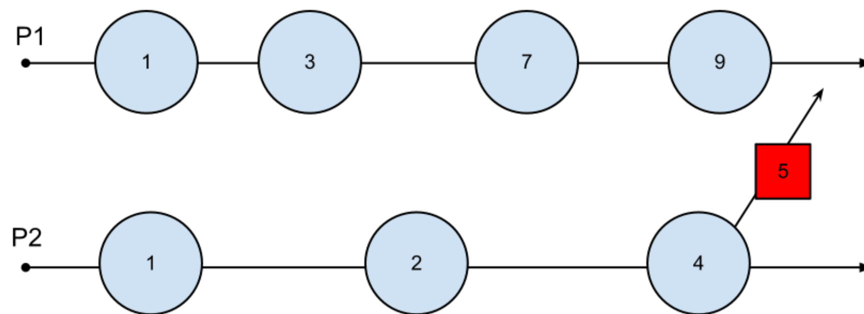

Figura 2.3: Mensagem *straggler*

Conforme ilustrado na figura 2.3, o processo p_1, cujo $LVT = 9$, recebe uma mensagem com o *timestamp* $t = 5$, ou seja, uma mensagem para ser tratada em um tempo virtual inferior ao seu atual LVT, o que caracteriza esta mensagem como uma mensagem *straggler*. Neste caso, o processo p_1 deve retornar para o $LVT = 5$ para tratar a mensagem recebida. Para isto, os eventos que foram tratados dentro da janela de retorno, ou seja, os eventos cujo LVT são $t = 7$ e $t = 9$ devem ser desfeitos, e novamente tratados após a inserção da mensagem recebida.

Um *rollback* causado pelo recebimento de uma mensagem *straggler* é chamado de *rollback* primário.

Caso o processo lógico tenha enviado alguma mensagem de dentro da janela de retorno, estas mensagens devem ser canceladas pelo processo durante o *rollback*. Isto é necessário porque uma mensagem enviada de dentro da janela de retorno faz parte da anomalia de causa e efeito do sistema.

Uma vez que cada mensagem enviada contém dados não confiáveis (devido à anomalia de causa e efeito), todo processamento oriundo destas mensagens também deve ser desfeito. Para que isto seja feito, para cada mensagem enviada pelo processo lógico dentro da janela de retorno, uma mensagem de cancelamento deve ser enviada, indicando a invalidação da mensagem original. Esta mensagem de cancelamento é denominada anti-mensagem.

A figura 2.4 ilustra uma situação semelhante ao representado pela figura 2.3. Neste caso, além de desfazer os eventos e_7 e e_9, o processo em questão deve cancelar a mensagem enviada no tempo $t = 7$ (cujo *timestamp* é $t = 12$) para o processo p_1. Isto é feito enviando uma anti-mensagem ao processo p_1, indicando que este deve descartar a mensagem recebida (e todo processamento relacionado à ela), por se tratar de uma mensagem que fere a consistência do sistema.

Quando um processo lógico recebe uma anti-mensagem ele verifica se o evento relacionado à esta já foi tratado ou está na lista de eventos futuros. Caso o evento esteja na lista de eventos futuros, este simplesmente é descartado (removido da lista).

Caso o evento já tenha sido tratado, existe a necessidade de se desfazer a simulação e restaurar o processo para um tempo inferior ao *timestamp* deste evento, e então descartar o evento em questão. Neste caso, o processo que recebeu a anti-mensagem deverá realizar um *rollback* para um LVT que garanta a consistência da simulação (ou seja, para um momento anterior ao processamento da mensagem recebida), desfazendo o tratamento do evento citado. Este *rollback* realizado devido ao recebimento de uma anti-mensagem é denominado *rollback* secundário.

Novamente, existe a possibilidade de ter havido envio de mensagem nesta nova janela de retorno e o procedimento de envio de anti-mensagens referente a cada mensagem enviada deve ser realizado, a fim de neutralizar todas as mensagens enviadas decorrentes de um processamento não-consistente.

Todo o processo de envio de anti-mensagens ocorre até que o sistema se encontre em um ponto onde não haja mais inconsistência. A partir deste ponto a simulação é retomada.

Uma desvantagem de um protocolo como o *Time Warp* é a existência de *rollbacks* secundários, que por sua vez podem disparar mais *rollbacks*, degradando o desempenho do sistema. Assim, um protocolo que não utiliza anti-mensagens pode ser bastante útil. O surgimento do protocolo *Rollback* Solidário, proposto

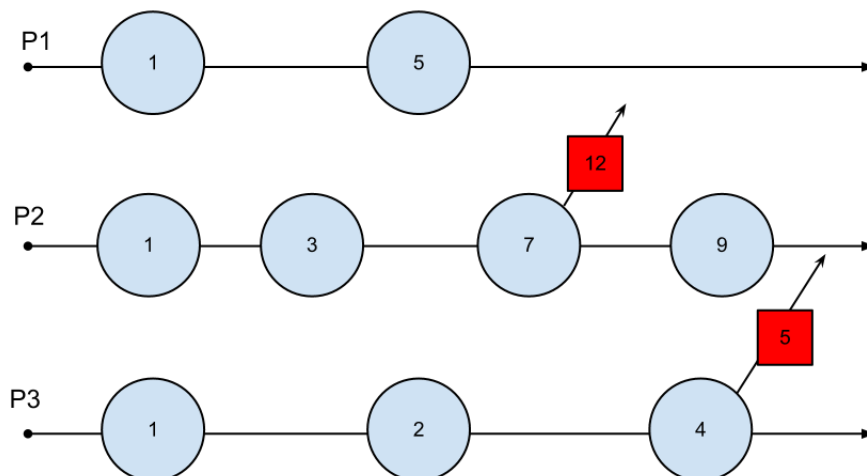

Figura 2.4: Envio de anti-mensagem para cancelamento de mensagem

por Moreira (2003), apresenta uma solução utilizando o conceito de *checkpoints* consistentes para recuperação do sistema, possibilitando que todos os processos lógicos executem um retorno ao mesmo tempo.

2.4 Balanceamento de cargas

Um item que deve ser considerado ao se distribuir uma simulação entre vários nós de um sistema é o balanceamento das suas cargas. Uma simulação somente aproveitará ao máximo a capacidade de processamento do sistema se os seus processos lógicos estiverem devidamente distribuídos por entre os nós do sistema, não causando sobrecarga em um determinado nó ou deixando nós ociosos.

Segundo Sivanandam e Visalakshi (2009), os algoritmos de balanceamento de cargas são desenvolvidos para dividir a carga entre os processadores e maximizar sua utilização enquanto minimiza o tempo final de execução das tarefas. Para que essas metas sejam atingidas, o algoritmo deve ser capaz de dividir as cargas de maneira que nenhum nó fique ocioso ou sobrecarregado. Ainda segundo Sivanandam e Visalakshi (2009), existe também a necessidade de se minimizar o custo de

comunicação entre os diversos processos lógicos no sistema.

Para sustentar a migração de processos lógicos durante a simulação, a fim de promover o balanceamento de carga, é proposto neste trabalho o desenvolvimento de um *middleware* de comunicação que encapsule as funcionalidades de migração e troca de mensagens, permitindo a utilização de tais mecanismos de maneira totalmente transparente. Um fator chave proposto por este *middleware* de comunicação é a continuidade da comunicação entre os processos lógicos através de troca de mensagens de maneira transparente, mesmo após uma possível migração do referido processo lógico para um *environment* diferente do seu original.

2.4.1 Escalonamento de processos em um sistema distribuído

Balancear a carga em um sistema distribuído consiste em distribuir os processos lógicos por entre os ambientes do sistema de simulação, de forma que as cargas desses ambientes estejam equilibradas. Essa alocação pode ser feita apenas no início da execução da simulação ou através de migração de processos entre os ambientes durante a simulação.

Os objetivos dos algoritmos de distribuição de carga são: maximização da velocidade de execução e nivelamento da carga entre os diversos nós do sistema de simulação.

A carga de um sistema é medida através de um índice de carga. Um índice de carga mede a utilização de algum elemento do sistema (BRANCO, 2004). Diversos índices de carga podem ser usados para o balanceamento. Cabe ao desenvolvedor do sistema de simulação distribuída definir quais índices serão levados em conta na aferição da utilização do sistema.

Alguns exemplos de dados que podem ser relevantes para se aferir a distribuição de carga no sistema são: tempo ocioso do processador, tráfego de mensagens entre os nós do sistema, quantidade de processos lógicos em cada nó do sistema, tamanho

das filas de eventos futuros em um determinado processo lógico, entre outros.

Parâmetros como o tempo ocioso de um processador ou a taxa de troca de mensagens entre processos lógicos são dados que podem ser utilizados para se prover um balancemento de carga em aplicações distribuídas em geral. Já dados como o tamanho da fila de eventos futuros de um determinado processo lógico, por exemplo, são intrinsicamente ligados à simulação distribuída.

Os algoritmos desenvolvidos para promover o balanceamento de carga em sistemas de simulação distribuída, utilizando informações sobre parâmetros da simulação, apresentam melhores desempenhos que aqueles desenvolvidos para aplicações paralelas em um âmbito geral (VOORSLUYS, 2006).

2.4.2 Migração de processos

Os algoritmos de escalonamento utilizam a migração dos processos lógicos a fim de equilibrar a distribuição da carga no sistema. Isto permite que os processos sejam realocados em tempo de execução, possibilitando assim um balancemento dinâmico de cargas.

Os objetivos da migração de processos são: (1) balancear dinamicamente a carga do sistema, (2) aproximar os processos dos recursos que acessam, (3) permitir tolerância a falhas, (4) melhorar o desempenho da comunicação entre os processos e (5) melhorar o processo de administração do sistema (MILOJICIC et al., 2000).

Migrar um processo em execução significa transferir seu estado entre dois computadores, permitindo que a execução do processo possa continuar no computador de destino do mesmo ponto em que foi suspensa no computador de origem. Assim que for determinado o processo que sofrerá migração, o mecanismo de migração de processos é responsável por suspender a execução do processo no computador de origem, enviar o estado do processo para o computador de destino e reiniciar a execução do processo.

2.4.3 O Uso de Agentes Móveis Para Prover Mobilidade

Agentes Móveis são entidades computacionais com características como autonomia, habilidades sociais, capacidade de aprendizado e, mais importante, mobilidade (LANGE; OSHIMA, 1999). Um agente móvel é um processo que possui a capacidade de transportar seu atual estado de um ambiente para outro, incluindo o estado de suas variáveis internas.

Esta característica de mobilidade oferecida pelos agentes móveis supre, de certa forma, uma das necessidades no desenvolvimento de aplicações de simulação distribuída que pretendem prover, de alguma forma, balanceamento de cargas entre os nós do sistema.

Aproveitando a mobilidade de um agente móvel, Ribeiro, Walbon e Takahashi (2009) desenvolveram uma implementação do protocolo *Rollback* Solidário utilizando a biblioteca *Aglets* (TAI; KOSAKA, 1999) de agentes móveis.

Outras soluções como Perrone et al. (2006), Kuhlman et al. (2011) e Madireddy e Kumara (2011) baseiam-se nos agentes móveis para desenvolver a aplicação de simulação distribuída, porém não necessariamente aproveitando da mobilidade proporcionada por eles para prover balanceamento de carga.

Em Chan e Son (2010), os autores utilizam agentes móveis para apresentar modelos de sistemas e a simulação de seu comportamento, utilizando como exemplos modelos de redes sociais, biologia, química, ciência dos materiais, entre outros. Os autores exploram um ambiente virtual (denominado por eles de *environment*) onde os agentes interagem, e essa mobilidade influencia na simulação. A mesma técnica é sugerida por Railsback, Lytinen e Jackson (2012).

A utilização de agentes móveis para representar a mobilidade de um objeto em um determinado ambiente é utilizada por Middleton (2010). Em seu trabalho, os agentes móveis são empregados na criação do modelo a ser simulado e a sua mobilidade é analisada pelo algoritmo de simulação. Com isso, é possível, aferindo a mobilidade do agente dentro do ambiente criado, extrair dados referentes ao seu

comportamento.

Diferente do proposto por Chan e Son (2010), por Middleton (2010) e por Perrone et al. (2006), a mobilidade dos processos lógicos na arquitetura proposta neste trabalho visa prover o balanceamento de carga no sistema de simulação distribuída. No modelo aqui discutido, a localização do processo lógico (em qual *environment* ele executa) não influencia no resultado da simulação, mas sim no tempo necessário para que esta ocorra.

2.4.4 Requisitos Para Mobilidade de um Processo Lógico

A utilização de agentes móveis no desenvolvimento de uma simulação distribuída é justificada quando se deseja aproveitar a mobilidade oferecida pelo agente. Uma alternativa ao uso de agentes móveis é a criação de entidades que encapsulem os processos lógicos e supram suas necessidades básicas para o desenvolvimento da aplicação de simulação distribuída: autonomia, migração e comunicação entre os processos.

Para isto é criada uma abstração na qual os processos lógicos habitam. Essa abstração é denominada ambiente (*environment*). Todo processo lógico habita um *environment*, e todo processo lógico que habita um determinado ambiente possui a capacidade de migrar do ambiente em que este se encontra atualmente para um novo ambiente, e lá continuar seu processamento.

Sendo assim, existe a capacidade de manipular a localização dos processos lógicos ao longo da simulação. Isso possibilita que a localização dos processos lógicos seja rearranjada conforme a necessidade da simulação ao longo do tempo, a fim de se alcançar um melhor aproveitamento dos recursos existentes, provendo o balancemento das cargas no sistema.

O *middleware* de comunicação proposto por este trabalho contempla a criação desta abstração de ambiente onde os processos lógicos residem, assim como provê uma base para a criação destes processos lógicos, fornecendo a eles a capacidade

de comunicação através de troca de mensagens e de migração por entre os diversos *environments* existentes no sistema de simulação.

2.5 Utilização do middleware de comunicação para o desenvolvimento de um framework para simulação distribuída

Escrever uma aplicação de simulação de eventos discretos distribuída é uma tarefa complexa. Todo o tratamento de sincronização, comunicação, troca de mensagens, manipulação de objetos remotos, entre outras tarefas, acarretam no surgimento de diversos detalhes, alheios à simulação propriamente dita, que devem ser gerenciadas pelo desenvolvedor que pretende implementar a simulação.

Somam-se a isso questões de ordem prática, como o cuidado com o desempenho (entram neste item: balanceamento de carga, *design* eficiente dos algoritmos utilizados, etc) e permissividade ao erro (a probabilidade de se introduzir um erro em um código aumenta proporcionalmente ao tamanho deste código (ZHANG; TAN; MARCHESI, 2009)). Neste sentido obtém-se um cenário onde o desenvolvedor acaba tendo que se preocupar demasiadamente com detalhes alheios à simulação, tornando a simulação uma tarefa dispendiosa e altamente propensa a erros.

Assim como proposto em Cruz (2009), um *framework* de simulação tem o objetivo de suportar o desenvolvimento de simulações distribuídas de uma maneira que proveja encapsulamento dos mecanismos alheios à modelagem e execução da simulação, transparência nas tomadas de decisões internas e reusabilidade de código.

A opção por um *framework* acarreta uma série de vantagens por excluir de seu usuário a responsabilidade de gerenciar diversas tarefas internas, deixando-o focado apenas na criação do modelo a ser simulado. O *middleware* de comunicação proposto por este trabalho visa oferecer mobilidade aos processos lógicos utilizados

pelo *framework* de simulação distribuída provendo três funcionalidades principais: encapsulamento, transparência e reusabilidade.

2.5.1 Encapsulamento

Uma das funções do *middleware* de comunicação é encapsular diversos elementos que não tratam diretamente da simulação, porém sustentam funcionalidades que dão vida a esta. Exemplo disso é a possibilidade de se isolar os mecanismos de troca de mensagens e migração de processos lógicos. Ao usuário do *middleware* basta chamar as funções necessárias para, por exemplo, migrar um determinado processo lógico de seu *environment* atual para um novo. Cabe ao *environment* o trabalho de suspender a execução do processo em seu ambiente local, enviar o processo para o seu novo ambiente e atualizar os dados referentes ao seu novo endereço físico, de forma que todas as mensagens enviadas à este processo lógico chegue agora em seu novo ambiente.

2.5.2 Transparência

A proposta de se escrever um código que seja ao mesmo tempo fácil de se implementar pelo usuário do *middleware* e eficiente em sua execução projeta-se diretamente na utilização de diversas camadas que ao mesmo tempo esconde do usuário algumas decisões internas e provê abstrações nas quais o usuário se apóia para desenvolver seu modelo.

Segundo Riehle (2000), um usuário ao utilizar um *framework* reutiliza seu *design* e sua implementação. Isto é feito pois cabe ao *framework* resolver os problemas referentes ao seu domínio. No caso proposto por esse trabalho cabe, de maneira análoga, a abstração por parte do *middleware* de comunicação toda a tomada de decisão relativa à comunicação e mobilidade, deixando ao usuário apenas a função de utilizá-lo, sem a necessidade de se preocupar com questões que estão fora de seu domínio.

2.5.3 Reusabilidade

Ao se eleborar um *middleware*, o responsável pelo seu *design* deve permitir que componentes internos deste sejam trocados ou mesmo customizados. Isso garante que o mesmo código escrito para ser executado em uma determinada aplicação continue a funcionar mesmo depois da troca de alguns de seus componentes internos.

2.6 Soluções Existentes

Uma proposta de *framework* para simulação distribuída foi apresentada por Cruz (2009), baseada em troca de mensagens suportando tanto *MPI* quanto *PVM* e abordando tanto os protocolos de sincronização *Rollback* Solidário e *Time Warp*. Em Ribeiro, Walbon e Takahashi (2009) é proposto uma solução utilizando agentes móveis, o que contempla, além da comunicação por troca de mensagens, também a possibilidade de migrações de processos lógicos através dos nós do sistema distribuído, visando a possibilidade de balancear as cargas no sistema.

Algumas propostas como a *Remote Call Framework (RCF)*, *ClassdescMP* e diversas implementações do *MPI* e de *PVM* proveem soluções para a troca de mensagem entre diferentes processos. Essas soluções oferecem eficientes mecanismos para gerenciar a troca de mensagens, porém não possuem soluções nativas para a migração de processos lógicos entre nós do sistema de simulação, e também não estão preparadas para o redirecionamento de mensagens enviadas aos processos que migraram para um nó diferente dos seus nós de origem.

Outras soluções como Perrone et al. (2006) e Kuhlman et al. (2011), assim como Ribeiro, Walbon e Takahashi (2009), baseiam-se nos agentes móveis para se desenvolver a aplicação de simulação distribuída. As soluções baseadas em agentes móveis proporcionam uma mobilidade aos processos lógicos, que é útil ao balanceamento de cargas. Porém, assim como as bibliotecas de comunicação

citadas anteriormente, os agentes móveis não estão preparados para redirecionar as mensagens destinadas aos processos que migraram de seus nós de origem.

2.7 Considerações finais

A mobilidade de agentes móveis é uma ferramenta de grande utilidade quando se deseja prover mobilidade dos processos lógicos em um sistema de simulação distribuída. Provendo mobilidade aos processos lógicos, ganha-se a possibilidade de se balancear a carga por entre os nós de um sistema de simulação.

Este trabalho propõe um modelo que oferece aos processo lógicos a mesma mobilidade provida pelos agentes móveis, com uma flexibilidade no redirecionamento de mensagens enviadas para processos lógicos que migraram de seu ambiente de origem. Tal funcionalidade é útil quando se deseja manter uma uniformidade na troca de mensagens após a migração de processos lógicos (o que pode ocorrer durante o balanceamento dinâmico de cargas).

3 O Projeto do Middleware proposto

Um *framework* de simulação distribuída tem como finalidade prover um mecanismo para o seu usuário descrever e executar uma simulação distribuída. Cabe ao *framework* se apoiar em uma camada de comunicação para poder oferecer, de forma transparente, a comunicação entre os processos lógicos e sua eventual migração. Visando isso, este trabalho propõe uma solução, no formato de um *middleware* de comunicação, para o desenvolvimento de um *framework* de simulações.

Conforme descrito na seção 2.6, os mecanismos de troca de mensagens existentes não provêem uma camada de abstração do endereço de um componente após uma eventual migração. Isto significa que ao se migrar um componente de seu nó de origem para um novo ambiente deve-se, de maneira explícita, informar a todos os componentes do sistema que o seu endereço físico mudou, para que as mensagens direcionadas a este processo lógico o alcancem em seu novo ambiente de execução.

Uma característica fundamental do *middleware* de comunicação proposto por esse trabalho é que, uma vez que um objeto tenha migrado de seu nó de origem para um novo local, o *middleware* se encarrega de redirecionar as mensagens destinadas a esse processo lógico em seu novo ambiente, deixando completamente transparente para o seu usuário questões como endereço físico do processo lógico, *status* do processo, etc.

3.1 Arquitetura e separação das camadas

A figura 3.1 apresenta uma visão ampla da arquitetura de um *framework* de simulação distribuída.

A camada superior, denominada aplicação, é a interface pela qual o usuário do sistema descreve o seu modelo a ser simulado. Cabe ao *framework* prover uma *API* com a qual o usuário descreverá o comportamento do seu modelo.

A camada intermediária compreende tanto os algoritmos responsáveis pelo gerenciamento da simulação (o *kernel* do *framework*) quanto os mecanismos de sincronização e balanceamento de carga. É nesta camada também que se encontra as descrições dos componentes elementares utilizados para a descrição do modelo. São exemplos de componentes: fila de eventos futuros, gerador de eventos e o processo lógico, responsável por consumir os eventos discretos.

Por fim, sustentando as demais camadas encontra-se o *middleware* de comunicação proposto neste trabalho. Esta camada é responsável não somente pela troca de mensagens entre processos lógicos, como também por todo o gerenciamento do ciclo de vida de um processo, serialização e migração de processos lógicos, redirecionamento de mensagens, gerenciamento de recursos do sistema e mecanismo de comunicação grupal.

3.2 Módulos do middleware

A arquitetura do *middleware* é dividida em dois módulos principais: o ambiente (*environment*) e o processo lógico (*process*) (ambos ilustrados no diagrama da Figura 3.2). Um ambiente representa um nó físico do sistema de simulação distribuída, ou seja, é a representação lógica de um computador no sistema distribuído. Em uma simulação distribuída, tipicamente, cada nó físico da rede deve conter um único *environment*.

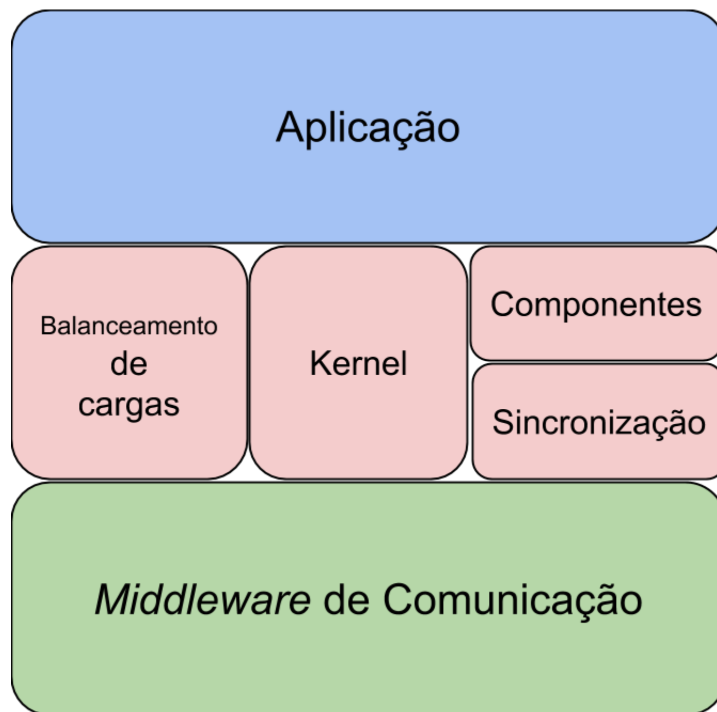

Figura 3.1: Camadas da arquitetura do *framework*.

O segundo módulo da estrutura do *middleware* é o processo lógico, que é a representação lógica de um processo no sistema de simulação distribuída. Na arquitetura aqui descrita, um processo lógico somente existe dentro de um *environment*. Sendo assim, um *environment* pode ser visto como um conjunto de processos lógicos. Por sua vez, um processo somente pode estar contido em um único *environment* em um determinado momento.

Assim é definido:

Definição 1: Um *environment* é um conjunto de processos.

Definição 2: Um processo só pode estar contido em um único ambiente em um determinado instante.

Tanto o *environment* quanto o processo são abstrações lógicas que representam a simulação baseada em eventos discretos. Fisicamente, tanto o ambiente quanto o processo lógico são instâncias de objetos que devem ser implementadas estendendo

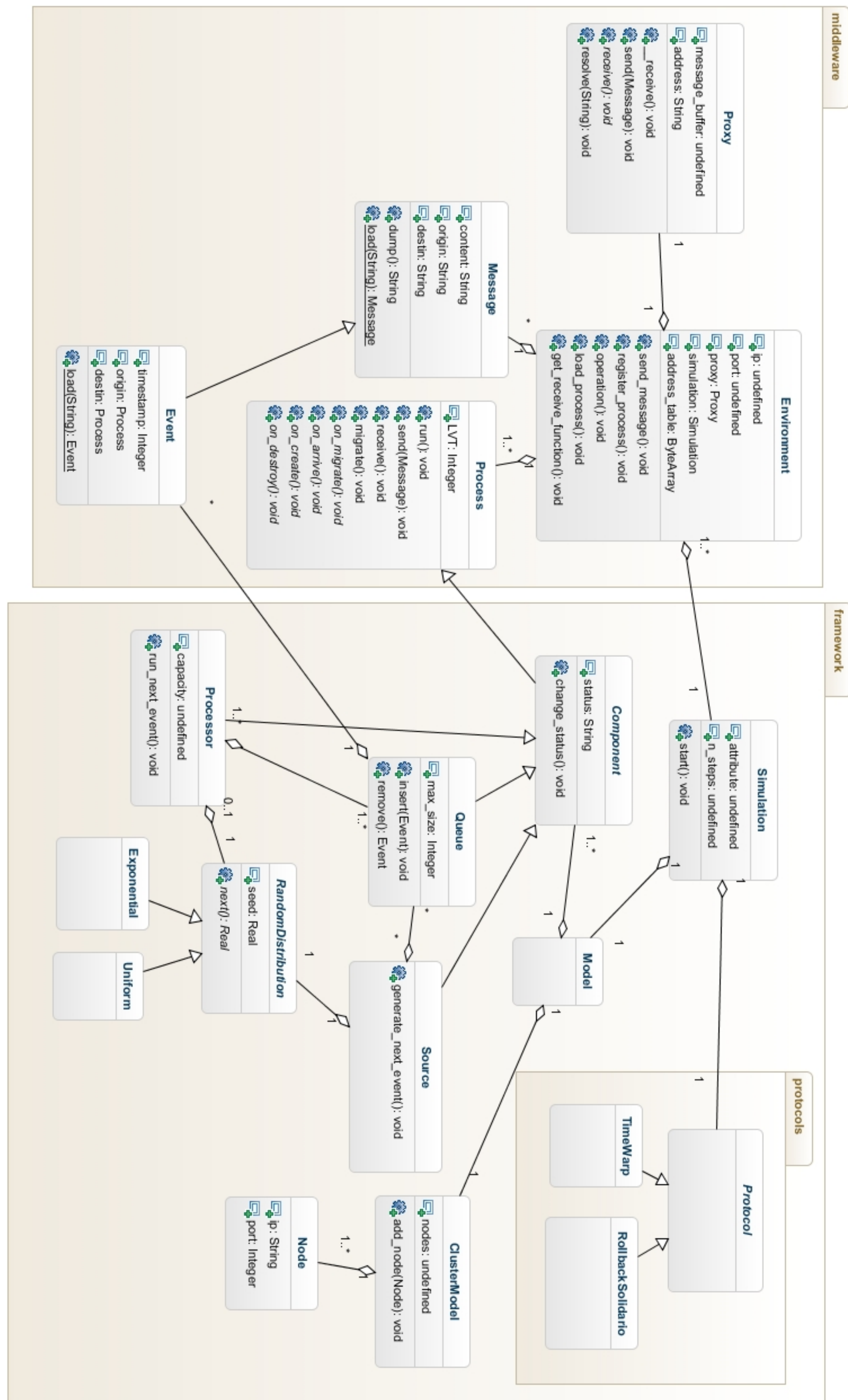

Figura 3.2: Diagrama de classes.

classes bases abstratas que contém as especificações descritas por este *middleware*.

A arquitetura do *middleware* aqui proposto deve oferecer as seguintes funcionalidades:

- Comunicação entre processos lógicos por troca de mensagens.
- Serialização do conteúdo de um processo lógico.
- Migração de um processo de um *environment* para outro.
- Continuidade da comunicação de maneira transparente, mesmo após a migração de um processo.
- Serialização em larga escala de um ambiente ou de toda a simulação.
- Comunicação grupal entre processos e entre ambientes.

A propriedade de comunicação por troca de mensagem é um item fundamental para a implementação de simulação distribuída. A forma de se comunicar por troca de mensagens provida pelo *middleware* baseia-se nas quatro suposições iniciais descritas por McQuillan e Walden (1975) sobre os canais de comunicação interprocessos:

- O canal introduz um flutuante, porém finito, atraso nas mensagens.
- O canal possui uma flutuante, porém finita, largura de banda.
- O canal apresenta uma flutuante, porém finita, taxa de erro.
- Existe uma real possibilidade de que as mensagens transmitidas da fonte para o destino cheguem ao destino em uma ordem diferente da originalmente transmitida. É assumido que tanto a fonte quanto o destino possuem, em geral, finitos tamanhos de *buffers* de armazenamento e diferentes *bandwidth*.

A capacidade de um processo lógico migrar de um *environment* para outro é um ponto fundamental para se viabilizar o balanceamento de cargas em uma simulação distribuída. O mecanismo de migração, descrito na Seção 3.5.3, é a união da capacidade de serialização de um processo e da comunicação entre diferentes *environments*, uma vez que a migração consiste em serializar o processo em seu ambiente de origem e transmiti-lo (em sua forma serializada) para um novo ambiente.

3.3 A classe Environment

Essencialmente, um *environment* na arquitetura aqui proposta é uma plataforma que abriga e gerencia diversos processos lógicos. A existência desta plataforma como base para o gerenciamento dos processos é justificada quando deseja-se manipular simultaneamente características de diversos processos que possuem em comum o fato de estarem no mesmo ambiente físico (como por exemplo migrar ou serializar todos os processos). Mas principalmente se justifica quando se deseja ter uma camada que seja responsável por gerenciar o ciclo de vida destes processos, como a criação, migração e destruição destes processos lógicos.

3.3.1 Estrutura interna

Internamente, o *environment* apresenta as seguintes estruturas básicas (conforme ilustrada na Figura 3.3):

- Estrutura interna de dados do ambiente;
- Tabela de endereços de processos;
- Lista de processos lógicos locais;
- *Proxy* de comunicação externa.

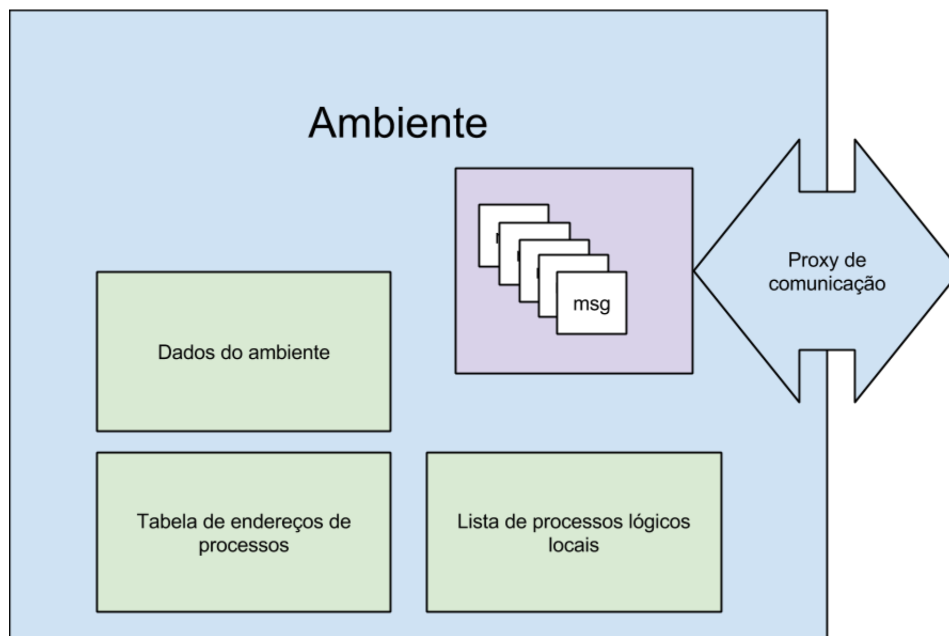

Figura 3.3: A arquitetura interna de um *environment*.

A estrutura interna de dados do ambiente é a representação de todos as variáveis pertencentes ao *environment*. Dados como o endereço físico (IP:PORT) do ambiente na rede, endereço lógico do ambiente e quantidade de processos residentes no *environment* são armazenadas neste espaço.

Um *environment* possui um endereço lógico único em todo o ciclo de vida da simulação, denominado *Unique Environment Identifier - UEI*. Este endereço é um número natural que o representa no sistema de simulação.

A comunicação entre dois *environments* se dá através de seus endereços físicos na rede (IP:PORT). Tal inflexibilidade é justificada quando se assume que um *environment* tem todo o seu ciclo de vida atrelado a um mesmo nó físico do sistema.

3.3.2 Tabela de endereços de processos

A tabela de endereços dos processos (atributo `address_table` da classe `Environment`, ilustrada no diagrama da figura 3.2) é uma lista associativa que possui

informações básicas de endereços e *status* dos processos existentes. Esta tabela contém informações não apenas dos processos residentes no *environment* em questão, mas também dados referentes aos processos residentes em outros ambientes de simulação pertencentes ao mesmo sistema. Esses dados são necessários para se prover a comunicação entre processos residentes em ambientes distintos.

Todo processo possui um endereço lógico e um endereço físico. O seu endereço lógico é único em toda a simulação, e não muda mesmo se o processo migrar de um ambiente para outro. Já o seu endereço físico depende do *environment* em que ele está em um determinado momento. A tradução de endereço lógico para endereço físico se faz utilizando a tabela de endereço de processos. Após a migração de um processo lógico, o *proxy* do ambiente que recebeu o processo em questão fica responsável por, através de envio de mensagens, atualizar o novo endereço físico do processo que migrou. Assim, os demais *proxies* de cada ambiente do sistema atualizam suas tabelas de endereços, a fim de obter o endereço correto do processo em questão.

Além de armazenar dados referentes aos endereços lógicos e físicos dos processos, a tabela de endereços armazena também uma variável referente ao *status* de cada processo. Os estados de um processo podem ser:

- Ativo: Indica que o processo se encontra neste *environment* e está ativo.
- Ausente: Indica que o processo em questão se encontra em outro ambiente.
- Trânsito: Indica que o processo em questão estava em um momento anterior neste ambiente e migrou para um ambiente diferente, porém ainda não atualizou a tabela de endereços com seu endereço físico atual.
- Inativo: Indica que o processo se encontra neste ambiente, mas não está ativo.

Os estados ativo e inativo são os estados mais comuns de um processo em um sistema típico. Eles indicam que o processo em questão está, ou não, em

execução naquele ambiente. Um processo em estado inativo significa que este está residente no *environment* em questão, mas não está executando no momento por algum motivo não identificado. Toda mensagem recebida para ser entregue a um processo inativo será armazenada em um *buffer* de mensagens no *proxy* e deverá ser retirada posteriormente pelo processo.

O estado de trânsito indica que o processo esteve naquele *environment* em algum instante do passado e que sofreu uma migração, mas seu novo endereço não foi ainda atualizado. Mais informações de como funciona a atualização de endereços durante a migração podem ser encontradas na seção 3.5.4.

3.4 A classe Proxy

O *proxy* é a camada do *environment* que é responsável por toda troca de mensagens entre os processos. O *proxy* atua de maneira distinta em troca de mensagens entre processos que convivem no mesmo ambiente e entre trocas de mensagens entre processos que se situam em ambientes distintos. As distinções entre trocas de mensagens internas e externas serão tratadas na seção 3.5.5.

O *proxy* também é o único canal por onde os processos recebem mensagens vindas de fora do ambiente. Toda mensagem recebida pelo *proxy* é identificada pelo endereço lógico do processo destinatário. Cabe ao *proxy* converter este endereço lógico em seu endereço físico e encaminhar a mensagem ao destino. Isto garante que, mesmo após uma migração, um processo lógico continua acessível pelo mesmo endereço lógico.

Como todo o tratamento de envio e recebimento de mensagens deve ser não-bloqueante, o *proxy* deve operar de maneira assíncrona aos demais módulos do *environment*.

A listagem 3.1 traz uma implementação que ilustra o funcionamento de um *proxy* de comuicação. Esta implementação foi realizada utilizando a linguagem

Python, uma linguagem de alto nível orientada à objetos e que possibilita uma rápida prototipação, além de ser amplamente utilizada em computação científica.

Listing 3.1: Implementação mínima de um *proxy*

```
import socket
import thread
import time
STATUS = ['active', 'away', 'transit', 'inactive']

class Proxy(object):

    def __init__(self, ip, port, name=None):
        self.components = []
        self.ip = ip
        self.port = port
        thread.start_new_thread(self.__receiver, (ip,port))

    def __receiver(self, ip, port):
        HOST = ip      # Symbolic name meaning all available interfaces
        PORT = port    # Arbitrary non-privileged port

        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.bind((HOST, PORT))
        while 1:
            s.listen(1)
            conn, addr = s.accept()
            data = conn.recv(2048)
            if data:
                self.receive(data)
            else:
                conn.close()
```

```
    def register(self, component):
        pass

    def modify_component_status(self, component, status):
        self.components[component.id]['status'] = status

    def modify_component_address(self, component, addres):
        self.components[component.id]['addres'] = addres

    def receive(self, msg):
        """receive a message from the external world"""

    def send(self, msg, destin):
        """Send a message to external world"""
        HOST,PORT = self.resolve(destin)
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((HOST, PORT))
        s.sendall(msg)
        s.close()

    def resolve(self, destin):
        pass
```

Os métodos `resolve`, `register` e `receive` são abstratos, e devem ser implementados por quem estender a classe `Proxy`.

Ao se instanciar a classe `Proxy`, o seu construtor (o método `__init__`) dispara uma nova *thread* que executa o método `__receive`. Este método é o responsável por escutar o *socket* e receber as mensagens através da rede. Quando o *socket* recebe uma nova mensagem através da rede, o método `__receive` (que está execu-

tando em uma `thread` independente) repassa a mensagem para o *proxy* invocando o método `receive`, passando o conteúdo da mensagem como parâmetro.

O método `send` da classe `Proxy` é o responsável por enviar uma mensagem para um segundo *proxy*. Este método recebe a mensagem a ser enviada e o endereço do *proxy* de destino. O *proxy* utiliza o método `resolve` para converter o endereço lógico do destinatário para seu equivalente físico. Cabe ao usuário do *proxy* implementar tal método.

3.5 A classe Process

Um processo é uma unidade de processamento em um sistema de simulação de eventos discretos. É ele o responsável por retirar cada evento a ser executado da fila de eventos futuros e executá-lo. A representação de um processo lógico na arquitetura aqui descrita se dá pelo objeto `process`.

Um processo lógico possui duas identificações distintas no sistema: seu endereço físico em memória e seu endereço lógico no sistema de simulação. Na arquitetura deste *middleware*, o processo é referenciado em uma troca de mensagens sempre pelo seu endereço lógico. Cabe ao *proxy* do ambiente fazer, quando for necessário, a conversão do endereço lógico para seu equivalente endereço físico a fim de entregar a mensagem ao processo de destino.

Neste ponto, a comunicação se divide em dois tipos diferentes: comunicação entre processos cohabitantes (que habitam o mesmo *environment*) e comunicação de processos não-cohabitantes (*environment* diferentes). Essa diferença força a utilização de meios distintos de comunicação para cada caso, porém isto é resolvido internamente pelo *middleware*, deixando a comunicação transparente para o *framework*.

Mais detalhes sobre a comunicação entre processos e a distinção entre comunicação entre processos cohabitantes e não-cohabitantes são descritas na seção 3.5.5.

3.5.1 Ciclo de vida de um processo

Um processo lógico tal como descrito pelo *middleware* possui um ciclo de vida com fases bem definidas. Isso significa que a classe base a qual o usuário do *middleware* estende para criar seu processo lógico já prevê, na forma de métodos abstratos, espaços onde serão implementados trechos importantes para fases específicas da vida do processo. Essas fases estão ilustradas no diagrama da Figura 3.4. A representação dos *slots* onde serão inseridos os códigos na forma de métodos abstrados é ilustrada na representação UML da Figura 3.2.

O primeiro método invocado automaticamente pelo processo na sua criação é o método `on_create`. Este método é chamado apenas uma única vez em todo o ciclo de vida do processo lógico e trata das rotinas de inicialização do processo. Este método pode ser visto de maneira análoga ao construtor de uma classe na programação orientada à objetos. Imediatamente após executar o código contido no método `on_create`, o próximo método invocado automaticamente pelo sistema é o método `run`.

O método `run` é onde o corpo da simulação deve estar implementado. É neste ponto do código que, em um laço repetitivo, são retirados os eventos da fila de eventos futuros para a sua execução. Vale ressaltar que, como descreve Ribeiro, Walbon e Takahashi (2009), o processo de se retirar eventos da fila de eventos futuros para execução não pode ser feito na forma de um laço de repetição convencional, pois tornaria a execução do método não-preemptiva (bloqueando a execução neste laço).

Cabe aqui ressaltar também que a ação de retirar um evento da fila de eventos futuros deve ser implementada internamente na estrutura do *framework* de simulação, e o usuário do *framework* não precisa explicitamente descrever tal tarefa.

Os demais métodos invocados automaticamente pelo sistema são `on_migrate`, invocado antes da migração; `on_arrive`, invocado assim que o processo chega no destino; `hibernate`, invocado quando um processo é adormecido; e `on_destroy`,

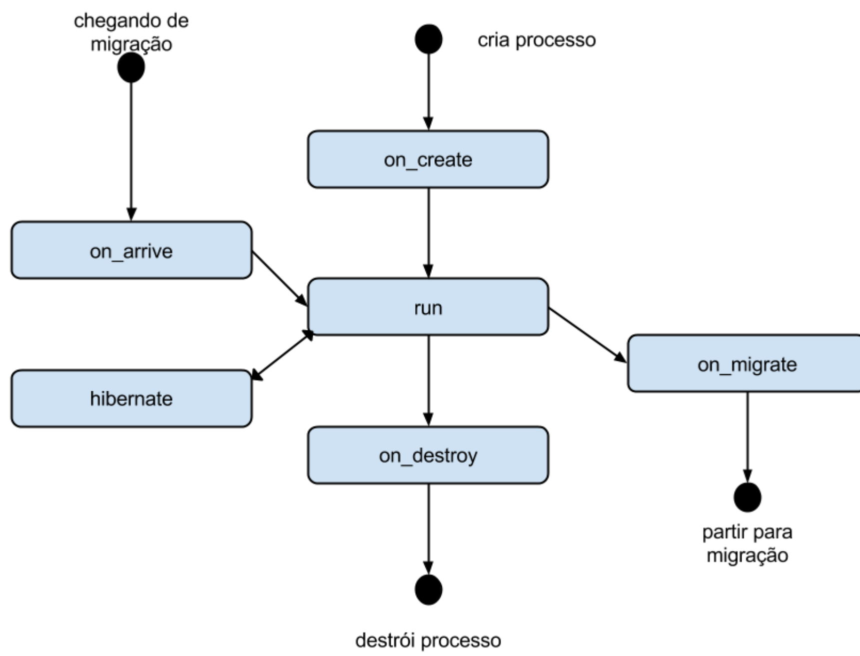

Figura 3.4: Ciclo de vida de um processo lógico.

invocado ao se destruir um processo lógico. O usuário do *middleware* é livre para estender tais métodos, adicionando o código que assim desejar para ser executado por estes.

Naturalmente, a classe `Process` comporta a criação de demais métodos além destes pré-estipulados. Porém, apenas estes métodos são executados de maneira automática pelo *middleware* em ocasiões especiais.

3.5.2 Serialização de um processo

Serialização é a capacidade que um objeto possui de converter sua estrutura interna de dados em uma representação binária ou codificada em texto. Tal ação é útil quando se deseja, por exemplo, persistir o estado de um objeto em memória.

Um processo deve permitir a serialização do seu conteúdo quando conveniente. A serialização de um processo é gerenciada internamente pelo *middleware* de comunicação. Cabe ao usuário garantir que todo o código escrito seja serializável.

3.5.3 Migração de processos

Uma das principais características propostas pelo *middleware* de comunicação que compõe este trabalho é a possibilidade de que processos lógicos migrem de seu nó de origem para um diferente nó do sistema a fim de, por exemplo, prover balanceamento de cargas. Para que isso ocorra, o nó destino deve possuir uma instância ativa de um *environment* capaz de gerenciar a continuidade da vida do processo em questão.

Para que a migração ocorra, uma série de ações devem acontecer em uma determinada ordem, garantindo a integridade do sistema durante este processo. A sequência de eventos para uma migração de um processo pode ser descrita como a seguir:

1. O *environment* recebe um pedido para migrar um determinado processo

lógico para um ambiente distinto.

2. O estado do processo (a ser migrado) na tabela de endereços do *environment* é modificado de `ativo` para `trânsito`.

3. O método `on_migrate` é invocado ainda no ambiente de origem do processo.

4. O processo a ser migrado é serializado pelo *environment* de origem e enviado para o ambiente de destino.

5. O ambiente de destino recebe o processo serializado e o desserializa, tornando-o um novo objeto em memória, mas mantendo os dados originais do processo.

6. O ambiente de destino atualiza o estado do processo em sua tabela local de Ausente para Inativo.

7. O ambiente de destino envia uma mensagem para o ambiente de origem indicando que o processo chegou, e qual o novo endereço físico do processo.

8. O *environment* de origem atualiza o estado do processo para `ausente`. Atualiza também o endereço físico do processo na tabela de endereços.

9. O processo, já migrado, executa o método `on_arrive` no ambiente de destino.

10. O *environment* muda o *status* do processo de `inativo` para `ativo`.

11. Por fim, o processo recupera todas as mensagens do *buffer* de mensagens do *proxy* de comunicação, resgatando eventuais mensagens recebidas enquanto o seu estado era `inativo`.

12. O processo, já em estado ativo, executa o método `run`.

13. Uma mensagem em *broadcast* é enviada a todos os *environments*, sinalizando o novo endereço físico correspondente ao processo que acaba de migrar. As tabelas de endereços são então atualizadas.

A figura 3.5 ilustra a ordem de eventos que ocorre no ambiente de origem de um processo durante uma migração. Assim que o environment recebe uma invocação do método `migrate` (com os devidos parâmetros que indiquem qual processo deverá migrar e qual o seu destino), este imediatamente invoca o método `change_status` do proxy, indicando que o processo em questão está em trânsito. Em seguida, é invocado o método `on_migrate` do processo a ser migrado. Este método é um método abstrato, reservado para o usuário do *middleware* adicionar o código que lhe convier.

O próximo passo é a chamada do método `send_process` pelo *environment*, que é responsável por serializar o processo lógico (utilizando o método `serialize`) e o enviar para o *proxy* de destino. Assim que o *environment* recebe a serialização, ele utiliza o método `send_message` para enviar o processo lógico (serializado) para o *proxy* de destino.

A figura 3.6 ilustra a ordem de eventos que ocorrem no ambiente de destino durante a chegada de um processo devido a uma migração. Assim que uma mensagem chega ao *proxy*, o método `receive` é automaticamente invocado. Em seguida o *proxy* invoca o método `load_process` do *environment* que é o encarregado de desserializar o processo lógico (através do método `load`). Em seguida o método `on_arrive` do processo lógico é chamado. Este é um método abstrato, que pode ser implementado pelo usuário do *middleware* caso seja conveniente.

Em seguida o *environment* utiliza o método `change_status` para mudar o estado do processo lógico de `ausente` para `ativo`. Por fim, o *environment* executa o método `run` do processo lógico, que é responsável por executar as tarefas pertinentes ao processo lógico (tratamento de eventos).

Assim que o processo termina o ciclo de ações de migração, ele está apto a continuar a simulação do ponto onde parou no ambiente antigo. Isto acontece porque, quando o processo foi serializado, todos os dados foram mantidos tais como estavam instantes antes da migração.

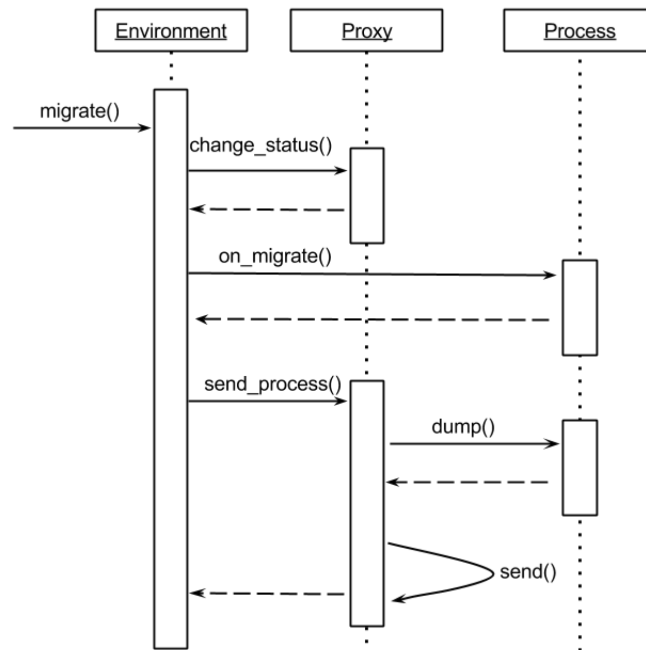

Figura 3.5: Diagrama de eventos durante uma migração no ambiente de origem

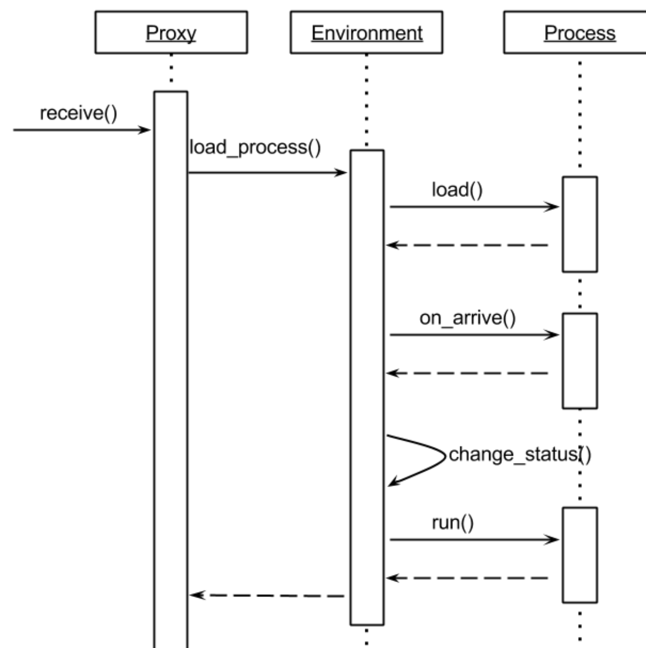

Figura 3.6: Diagrama de eventos durante uma migração no ambiente de destino

3.5.4 Atualização da tabela de endereços dos processos

Uma vez que um processo migra de um *environment* para outro, instantes após a migração, apenas os dois ambientes envolvidos na transação possuem os dados atualizados referentes ao processo que migrou. Todo ambiente diferente dos envolvidos no processo de migração possuem, em suas tabelas de endereços de processos, dados desatualizados quanto à sua localização, portanto, enviariam mensagens para o ambiente antigo, ao qual o processo em questão não mais pertence.

Ao receber uma mensagem destinada a um processo que não mais o habita, um *environment* (que possui o novo endereço físico do processo em questão), redireciona a mensagem ao *proxy* do ambiente que possui atualmente este processo. Isto acontece até que haja uma atualização de todas as tabelas de endereço, em todos os *environments* do sistema.

Em um determinado momento (programado para ocorrer periodicamente), uma ação de sincronismo de tabelas é disparada. Esta ação é iniciada de forma independente por cada *environment*, enviando uma mensagem para os demais ambientes, notificando-os de quais processos lógicos encontram-se em seu poder. Isto garante uma atualização constante das diversas tabelas de endereços de processos existentes na simulação.

3.5.5 Troca de mensagens

No modelo de comunicação implementado pelo *middleware* de comunicação, a troca de mensagens entre dois processos lógicos ocorre de maneira distinta, dependendo se os processos cohabitam um mesmo ambiente ou não.

Caso os processos residam no mesmo ambiente, é utilizada a chamada comunicação direta, onde a mensagem é diretamente transmitida de um processo para o seu destino. Porém, quando um processo deseja enviar uma mensagem a um

processo que habita um *environment* distinto, a mensagem é primeiramente encaminhada ao *proxy* do *environment* do processo remetente da mensagem, e cabe a este redirecionar a mensagem ao seu destino. A este processo dá-se o nome de comunicação indireta.

3.5.6 Comunicação local direta

A comunicação direta ocorre quando tanto o processo lógico que envia mensagem quanto o destinatário habitam um mesmo ambiente no sistema de simulação. Esta comunicação se dá através do endereço físico do processo destinatário, sem a necessidade de intervenção do *proxy* do ambiente para redirecionar a mensagem.

Quando o processo P_1 deseja enviar uma mensagem ao processo P_2, o processo remetente possui o endereço lógico do processo destinatário. O processo remetente então invoca um método interno de envio de mensagem que, por sua vez, contacta o *proxy* do sistema a fim de traduzir o endereço lógico em um endereço físico.

Ao receber um pedido de envio de mensagem, o *proxy* devolve ao processo P_1 uma função invocável que é responsável por enviar a mensagem ao destinatário. No caso da comunicação direta, a função devolvida pelo *proxy* possui o endereço físico em memória do processo destinatário (uma vez que o processo destinatário habita o mesmo ambiente, ou seja, o mesmo espaço de memória), e é capaz de enviar a mensagem diretamente para ele, sem uma nova intervenção do *proxy*.

A figura 3.7 ilustra a ordem de eventos executados ao se enviar uma mensagem utilizando comunicação direta. Quando o método `send` de um processo lógico é executado, este invoca o método `send_message` do *environment* ao qual ele está atrelado. O *environment* em questão irá verificar se o processo de destino da mensagem habita o mesmo ambiente (utilizando o método `is_local`, que verifica a tabela de processos lógicos). Ao ser confirmado que o processo lógico de destino habita o mesmo ambiente, o *environment* utiliza o método `get_receive_function` para receber a função `receive` do processo destinatário. Em seguida o *environment*

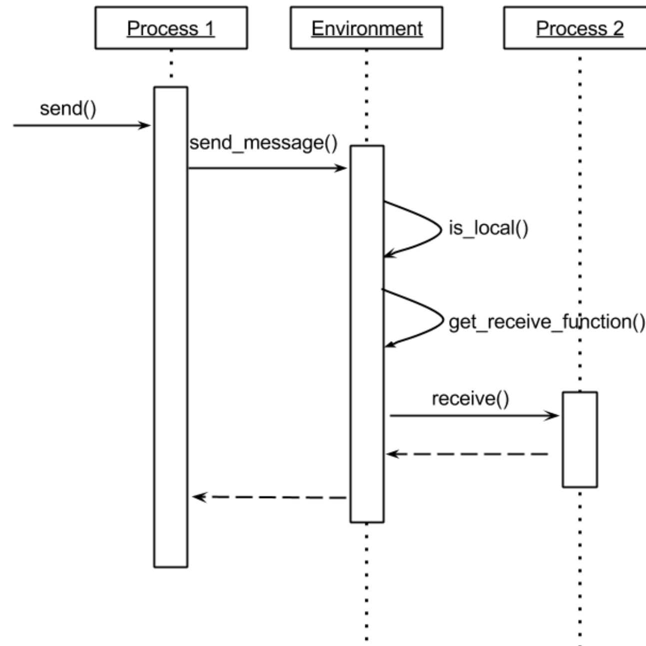

Figura 3.7: Diagrama de eventos durante uma comunicação local direta.

executa o método em questão, enviando a mensagem diretamente ao processo de destino.

Todas estas ações são realizadas internamente pelo *middleware* de comunicação e não necessitam de intervenção externa em momento algum.

3.5.7 Comunicação indireta

Quando a comunicação é feita por dois processos lógicos residentes em diferentes *environments* (e por consequência, em máquinas distintas da rede), a comunicação se dá através dos *proxies* dos *environments*.

O fluxo de comunicação é ilustrada de maneira simplificada na Figura 3.8.

Quando o processo P_1 deseja enviar uma mensagem a um processo P_3 que habita um ambiente distinto, ele procede da mesma maneira como descrito na seção 3.5.6: o processo remetente (P_1) invoca uma função de envio de mensagem,

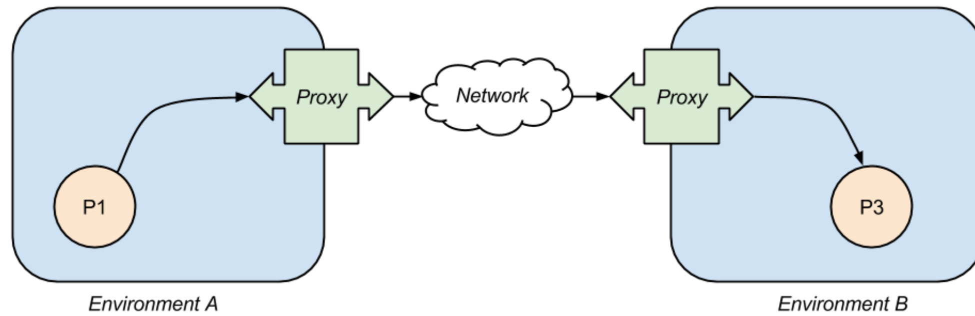

Figura 3.8: Comunicação indireta *proxy-process*.

passando o endereço lógico do processo destinatário (P_3). Esta função contacta o *proxy* de comunicação para fazer a tradução do endereço lógico do processo para o seu correspondente físico.

A partir deste momento, as ações ocorrem de maneiras distintas, pois P_3 habita outro ambiente. Neste caso, o *proxy* retorna para o processo P_1 uma função invocável que é responsável por armazenar a mensagem em um *buffer* interno do *proxy*.

Posteriormente, o *proxy* se encarrega de retirar todas as mensagens contidas neste *buffer*, e enviá-las aos *environments* correspondentes.

Assim que a mensagem chega ao *environment* de destino, cabe ao seu *proxy* de comunicação redirecionar as mensagens aos processos lógicos correspondentes.

A figura 3.9a ilustra a ordem de eventos ocorridos no ambiente remetente de uma mensagem via comunicação indireta. Assim que o método `send` do processo lógico é chamado, o processo invoca o método `send_message` do *environment* que ele habita a fim de tratar o envio da mensagem. O *environment* invoca então o método `is_local`, que indica que o processo destinatário habita em um ambiente distinto. Assim, o *environment* invoca o método `send_message` do *proxy* de comunicação para encaminhar a mensagem para um ambiente externo. O *proxy* utiliza o método `resolve`, que recebe o endereço lógico do processo em questão e devolve o endereço físico na rede (IP:PORT) do *proxy* responsável por reencaminhar a

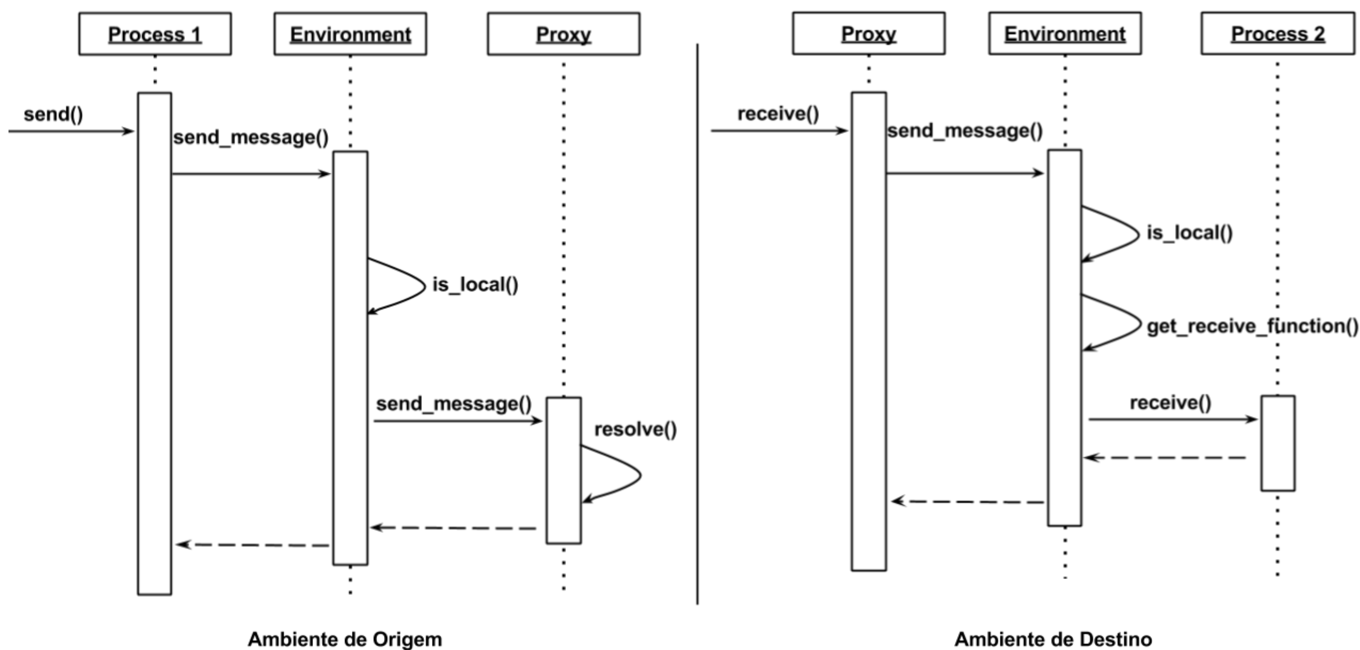

Figura 3.9: Diagrama de eventos nos ambiente de origem e de destino durante uma uma comunicação indireta.

mensagem.

A figura 3.9b ilustra a ordem de eventos ocorridos no ambiente destinatário ao se receber uma mensagem via comunicação indireta. Ao receber uma mensagem, o método `receive` do *proxy* é automaticamente invocada. Em seguida o *proxy* invoca o método `send_message` do *environment* para encaminhar a mensagem. O *environment* verifica se o processo destinatário o habita (utilizando o método `is_local`). Ao se confirmar que o processo em questão habita o ambiente local, o *environment* utiliza o método `get_receive_function` que retorna uma função invocável, denominada `receive`, que é responsável por enviar diretamente uma mensagem para o processo destinatário. Por fim, o *environment* invoca tal método, entregando a mensagem ao processo de destino.

A adoção da interferência dos *proxies* na transmissão das mensagens entre os processos pode parecer a princípio um item custoso e dispensável no sistema, uma vez que cada processo poderia abrigar sua própria tabela de endereços e fazer a

tradução de endereços físicos para lógicos de maneira direta. Porém, isso elevaria a quantidade de tabelas a se atualizar em caso de migração de processos.

Em um modelo onde cada processo resolveria independentemente a tradução de endereços lógicos para o seu correspondente físico para o envio de mensagens, a quantidade de tabelas de endereços seria proporcional à quantidade de processos lógicos existentes no sistema (e não proporcional à quantidade de ambientes, como é na arquitetura aqui proposta). Isso elevaria consideravelmente a quantidade de tabelas de endereços a serem atualizadas no caso de uma possível migração.

3.5.8 Migração e Comunicação Contínua

Ao se usar a comunicação indireta, ou seja, ao se referenciar um processo pelo seu endereço lógico, é introduzida uma transparência no envio de mensagem de um processo para outro. Neste cenário, o processo remetente não tem que se ocupar com detalhes referentes à transmissão das mensagens.

Quando um processo é movido de um ambiente para outro, este é serializado em seu ambiente de origem, enviado (através dos *proxies*) do seu ambiente inicial para o ambiente de destino e, já em seu novo ambiente, este processo é desserializado e inserido no contexto do sistema.

Quando isto é feito, o endereço físico do processo em questão mudou. Além disso, mudou também o ambiente onde este processo reside. Sendo assim, cabe ao ambiente que recebeu o processo disparar uma mensagem à todos os demais ambientes do sistema atualizando suas devidas tabelas de endereços dos processos com o novo endereço do processo que migrou.

Porém, mensagens podem ter sido enviadas ao processo durante o processo de migração. No caso de estas mensagens chegarem ao ambiente onde o processo não mais habita, há a necessidade de se encaminhar a mensagem para o novo ambiente responsável pelo processo em questão.

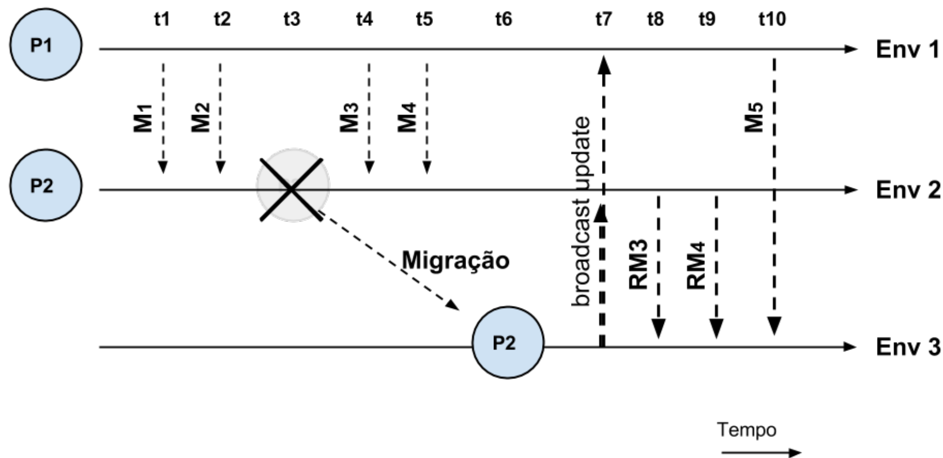

Figura 3.10: Diagrama de repasse de mensagens.

Na Figura 3.10 é ilustrado como as mensagens são repassadas durante uma migração de processos. Inicialmente, o processo P_1 habita o ambiente Env_1 e o processo P_2 habita o ambiente Env_2. Durante os tempos t_1 e t_2 as mensagens M_1 e M_2 são enviadas de P_1 para P_2. No tempo lógico t_3 o processo lógico P_2 inicia o processo de migração para o ambiente Env_3. A partir deste momento, todas as mensagens que chegam ao ambiente Env_2 (como as mensagens M_3 e M_4 da figura 3.10) destinadas ao processo P_2 são armazenadas em um *buffer*, e serão tratadas posteriormente.

No tempo t_6 o processo lógico P_2 finaliza o processo de migração para o ambiente Env_3. O novo ambiente por sua vez, utilizando-se do *proxy* de comunicação, dispara uma mensagem em *broadcast* para todos os ambientes atualizando o endereço de P_2.

Assim, o ambiente Env_2 que armazenava as mensagens M_3 e M_4 redireciona-as para o ambiente Env_3 (RM_3 e RM_4), onde se encontra o processo P_2 no momento.

Por fim, todas demais mensagens destinadas ao processo P_2, como a mensagem M_5, são encaminhadas diretamente ao ambiente Env_3.

3.6 Considerações Finais

A arquitetura do *middleware* de comunicação discutida neste capítulo visa prover as funcionalidades básicas para a implementação de um *framework* de simulação de eventos discretos.

O *middleware* proposto pretende ser uma alternativa para a implementação da simulação distribuída provendo, além de mobilidade e comunicação por troca de mensagens, uma maneira transparente de redirecionar mensagens após a migração de um processo lógico.

Ao se prover dois métodos distintos para comunicação (direta e indireta), provê-se, também, uma forma otimizada de se enviar mensagens para processos lógicos que habitem um mesmo ambiente.

Por fim, o *middleware* oferece também uma alternativa para a comunicação grupal, uma ferramenta necessária para a implementação de algoritmos de sincronismo, como o *Rollback* Solidário.

4 O Projeto de um Framework de Simulação

Para ilustrar o *middleware* discutido no capítulo anterior, este capítulo apresenta o projeto de um *framework* de simulação, utilizando o *middleware* de comunicação como base.

Este *framework*, conforme ilustrado na figura 4.1, é composto por diversos subcomponentes (aqui denominados genericamente de módulos). Cada módulo que compõe o *framework* se liga ao módulo central do *framework*, denominado *kernel*. Este *kernel* é o responsável por coordenar a simulação aplicando sincronização e balanceamento de carga, além de gerenciar o ciclo de vida da simulação (quando começar, quando terminar, etc).

O *framework* provê também uma biblioteca de componentes, que é a interface pela qual o seu usuário descreve o modelo a ser simulado.

4.1 O Módulo Componente

A biblioteca de componentes é a interface provida pelo *framework* para que o seu usuário descreva o modelo a ser simulado. Os componentes providos pela biblioteca de componentes devem ser utilizados em conjunto para descrever o comportamento do modelo, de maneira análoga aos componentes eletrônicos, que juntos compõem um circuito para realizar uma determinada ação. Cada componente pos-

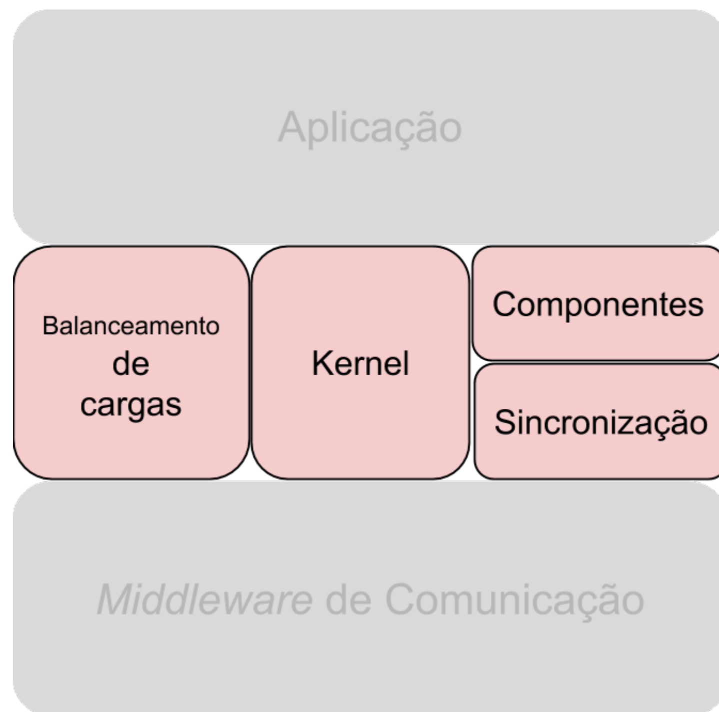

Figura 4.1: A camada aqui denominada *framework*.

sui suas características intrínsecas e características parametrizáveis por seu usuário. Estas características definem como este dado processo se comporta quando recebe um novo evento durante a simulação.

Um exemplo que ilustra a divisão do modelo em componentes discretos é a representação de uma fila de supermercado, por exemplo. Se enxergar cada cliente que vai ao caixa do supermercado como um evento discreto, pode-se considerar que o caixa é um componente que consome eventos. A fila, por sua vez, é um componente que armazena eventos e um terceiro componente (que não possui representação física direta) seria o gerador de eventos discretos, que insere novos eventos (clientes) na fila.

Cada um dos três componentes descritos possuem características intrínsecas (a fila armazena eventos, o caixa consome eventos e o gerador gera eventos) e características parametrizáveis, como por exemplo a taxa de criação de eventos

(quantos clientes entrariam na fila por minuto, obedecendo qual distribuição) ou a taxa de consumo de eventos (o quão rápido é o caixa).

Uma biblioteca de componentes deve proporcionar objetos parametrizáveis que permitam a descrição do comportamento de cada um deles. No caso citado anteriormente, o componente gerador deve suportar parâmetros que, por exemplo, descreva a taxa de criação de novos eventos, e as características de cada evento criado.

Os componentes são construídos diretamente a partir do objeto *process* do *middleware* de comunicação. Isso garante ao componente criado, por herança direta, toda funcionalidade de comunicação com outros componentes. Desta maneira, basta ao usuário do *framework* descrever na modelagem qual a conexão que cada componente faz, que esta é reproduzida automaticamente pelo *framework*, independente se esses componentes cohabitam ou não o mesmo ambiente.

Em alguns casos é conveniente que componentes sejam combinados a fim de que um novo componente seja criado, aumentando a granularidade de um componente no modelo. Uma das justificativas seria, por exemplo, garantir que dois componentes se comportem como um único, evitando assim que estes sejam por ventura separados e passem a cohabitar ambientes diferentes.

4.2 Componentes Básicos

O tipo primitivo de um componente no *framework* proposto é desenvolvido sobre a classe *process* do *middleware* de comunicação. Isso faz com que cada componente seja, por herança, um processo lógico no sistema de simulação.

Sobre este componente primitivo são construídos quatro componentes básicos (Figura 4.2), suficientes para demonstrar a implementação de alguns modelos para simulação. Estes componentes são: fila, gerador, consumidor e divisor.

Nem todos os componentes implementam esse modelo em sua totalidade. Com-

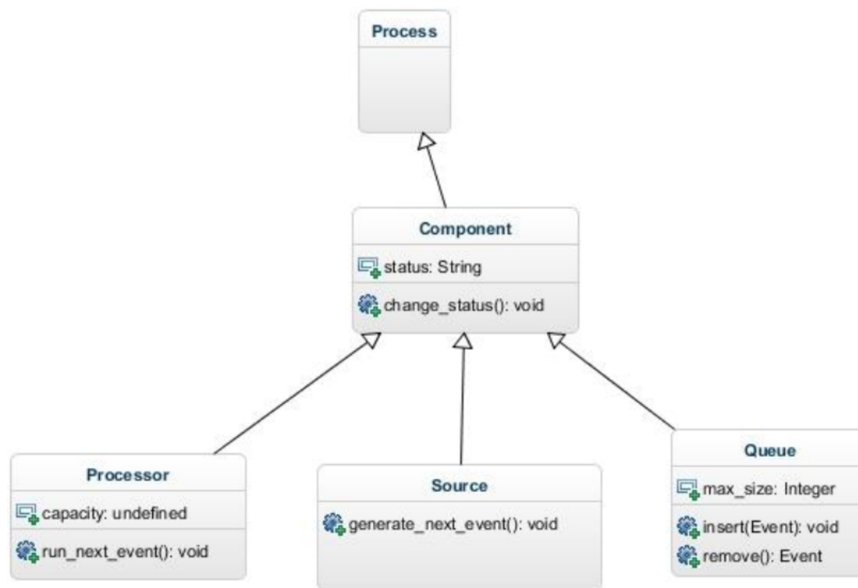

Figura 4.2: Hierarquia dos componentes básicos

ponentes como os geradores de eventos, por exemplo, não possuem canais de entrada de eventos, uma vez que a sua função é a de apenas gerar novos eventos com base em uma parametrização de suas características para que se comporte conforme o modelo matemático que descreve suas ações.

4.2.1 O Componente Fila

A fila é o componente responsável por armazenar eventos, ordenando-os com base no seu *timestamp*. Cada evento que chega à fila é alocado respeitando a ordem referente ao seu *timestamp*, e cada vez que um elemento é retirado da fila, isto é feito removendo o primeiro elemento da fila, ou seja, o elemento com o menor *timestamp*.

A fila é um componente passivo, ou seja, ela não executa nenhum tipo de ação sobre os eventos nela existentes. Por ser um componente passivo, é o componente gerador que deve executar a ação de inserir um novo evento na fila, e é um componente consumidor que deve executar a ação de retirar o evento de menor *timestamp*

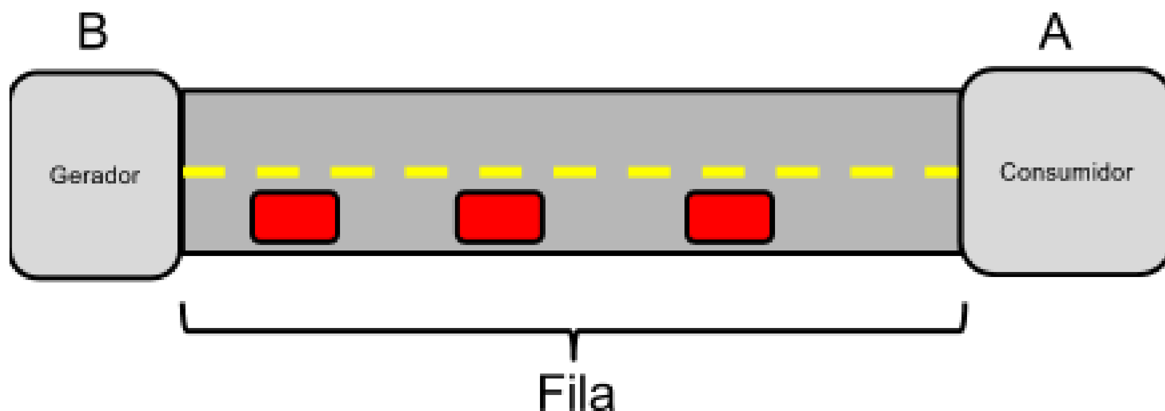

Figura 4.3: Modelagem de uma via urbana.

da fila.

Uma característica importante do componente fila é o seu tamanho. Uma fila pode ou não ter um tamanho definido, porém uma vez definido um tamanho para esta fila, quando este tamanho se excede, ou seja, quando a quantidade de eventos inseridos cresce em uma velocidade tão grande que o consumidor atrelado à fila não consegue consumi-los em tempo hábil, podem ocorrer duas situações, dependendo de como o usuário do *framework* descreveu em seu modelo. No caso mais simples, o usuário determina que ao se exceder a quantidade de elementos em uma fila, uma excessão do tipo `QueueOverflow()` seja lançada, e a simulação se encerra.

Em um caso mais elaborado, ao se atingir o limite de uma fila, esta pode apenas negar a inserção de novos elementos até que haja espaço suficiente. Neste caso, há uma propagação da condição de estacionamento da execução para os elementos atrelados à essa fila, causando um efeito que reflete em parte do sistema.

Um exemplo típico que ilustra esse caso é a simulação de uma sequência de cruzamentos em uma via urbana, ilustrada na Figura 4.3. A quantidade de carros que cabem em um intervalo entre os dois cruzamentos A e B é finito, e se o cruzamento A não consome carros em uma velocidade maior ou igual a que eles chegam, há um crescimento na quantidade de carros entre os dois cruzamentos, o

que pode levar à paralização temporária da simulação (eventos não podem mais ser criados), até que o consumidor A consuma eventos, liberando espaço na fila.

4.2.2 O Componente Gerador

Assim como ilustrado na Figura 4.3, um exemplo de gerador de eventos é uma extremidade de uma via de trânsito, que gera eventos (no exemplo citado, carros) em uma determinada taxa de tempo, com uma determinada frequência.

Um dos comportamentos parametrizáveis do componente gerador é a função que modela a taxa de criação de novos eventos ao longo do tempo. Um sistema em que se deseja maior fidelidade quanto aos dados simulados deve permitir uma detalhada parametrização do comportamento de seus componentes. Neste *framework*, isto é feito através das funções geradoras, que são funções que recebem, como parâmetros dados da simulação e resultam em decisões sobre a criação ou não de novos eventos, e o comportamento desses eventos.

4.2.3 O Componente Consumidor

Assim como o componente `gerador`, o componente `consumidor` é parametrizável através de funções externas que descrevem o seu comportamento ao longo do tempo de simulação. Outro dado levado em conta durante a execução de um evento por um consumidor são as características internas do evento. Eventos podem possuir parâmetros internos que influenciem no seu processamento, como por exemplo, designar se a sua execução cria um novo evento ou o tempo necessário para o seu processamento.

Uma vez que o evento é processado por um consumidor, este pode tomar três caminhos distintos: (1) o evento pode simplesmente ter seu ciclo de vida encerrado, (2) o evento pode ser redirecionado para um novo processador (ou mesmo para o mesmo processador, dependendo da necessidade da simulação), porém com suas características originais mantidas ou (3) o evento sofre uma modificação nas suas

características antes de ser encaminhado adiante.

4.3 O Kernel do Framework

A parte central do *framework* é desempenhada pelo módulo *kernel* (Figura 4.1). O *kernel* é o responsável por gerenciar a simulação, incrementando o *LVT* de cada processo lógico, além de tomar decisões de migração, comunicação, sincronismo e balanceamento de carga.

A cada incremento discreto do *timestamp*, o kernel do *framework* deve realizar as seguintes funções:

- Atualizar o *LVT* de cada processo lógico.
- Verificar se existem mensagens no *buffer* do *middleware*, e tentar redirecioná-las ao destinatário.
- Verificar, através do módulo de balanceamento de cargas, se existem processos que devem ser migrados.
- Verificar, através do módulo de sincronismo, se ocorreram mensagens *stragglers*. Caso afirmativo, designar ao módulo de sincronismo a tarefa de re-sincronizar o sistema.
- Gerenciar, através do módulo de sincronismo, as tarefas de salvamento de estados, cálculo do *GVT*, etc.

O *kernel* do sistema executa estes processos em *loop* até que o número de iterações previstas acabe, ou até que a simulação seja interrompida por algum evento externo.

4.4 Protocolos de Sincronização

Uma das premissas do projeto é a possibilidade de se substituir certos módulos do *framework* conforme a necessidade do seu usuário. Este encapsulamento de certas funcionalidades do *framework* oferece uma facilidade quando se deseja trocar algum componente do sistema.

Para que isto seja possível, o *kernel* do sistema deve garantir uma *interface* de acesso a diversas funções e variáveis do ambiente, permitindo assim que o módulo adquira dados referentes à simulação e interfira no seu funcionamento.

No caso da sincronização dos processos, o módulo em questão deve ser capaz de adquirir valores como o LVT e o GVT, identificar mensagens *stragglers*, além de ter acesso ao *middleware* de comunicação para trocar mensagens com os demais nós do sistema, e requerer o *rollback* para um determinado *timestamp*.

4.5 Algoritmos de balanceamento de Carga

Assim como no caso dos protocolos de sincronização de eventos, os algoritmos de balanceamento de cargas do sistema também são modulares e podem ser trocados pelo usuário (ou até mesmo customizados).

Cabe ao *framework* prover uma interface que possibilite ao módulo de balanceamento requerer ao *kernel* informações sobre a carga de cada nó do sistema, tal qual o perfil de comunicação de cada *process*. Com base nesses dados, o algoritmo decide qual processo deve ser migrado, e para qual máquina.

4.6 Considerações Finais

A arquitetura aqui discutida apresenta as características mínimas para a construção de um *framework* de simulação que utiliza componentes, na forma de classes

de uma linguagem orientada à objetos, para representar o modelo a ser simulado.

Esta decisão de *design* de *software* permite que se utilize das características da programação orientada à objetos a fim de facilitar a modelagem do sistema a ser simulado.

Características como extensão e herança de componentes permitem um encapsulamento do comportamento das partes do sistema, possibilitando o reuso de componentes, e facilitando o seu uso.

5 Exemplos de Uso

Este capítulo traz exemplos de utilização do *middleware* de comunicação e do *framework* de simulação propostos neste trabalho. Os trechos de códigos aqui discutidos têm como objetivo ilustrar, do ponto de vista do usuário, como utilizar o *middleware* de comunicação e o *framework* de simulação.

5.1 Troca de Mensagens

O primeiro exemplo tratado apresenta o funcionamento da troca de mensagens através do *middleware* de comunicação utilizando a classe `Environment`.

Neste exemplo, são instanciados dois ambientes distintos, e o primeiro ambiente envia, repetidamente, mensagens para um segundo ambiente. O ambiente emissor é descrito na listagem 5.1.

Listing 5.1: Ambiente Emissor

```
from T100 import Environment

env = Environment(ip="192.168.0.12", port=7777)

for i in range(10):
    env.send_message(i, destin=("192.168.0.15",7777))
```

O ambiente receptor, descrito na listagem 5.2, sobrescreve o método `receive`

criando uma funcionalidade para o *environment*. Neste caso, o método `receive` apenas imprime na tela a mensagem recebida.

Listing 5.2: Ambiente Receptor

```
from T100 import Environment

class Env(Environment):
    def receive(self, message):
        print message

env = Env(ip="192.168.0.15", port=7777)
```

5.2 Exemplo Básico de Modelagem

Este exemplo apresenta a ideia básica de se utilizar uma linguagem orientada à objetos para descrever um modelo a ser simulado. Neste caso, são criados três objetos, cada um representando um componente do modelo a ser simulado: uma fila de eventos (`q = Queue(...)`), uma fonte geradora de eventos (`s = Source(...)`) e um consumidor de eventos (`p = Process(...)`).

Listing 5.3: Exemplo básico de modelagem

```
from t100.core.simulator import Simulator
from t100.core.base_components import *

def creation_tax_expression(timestamp):
    # 2 events each five minut
    return 2.0/(60*5)

def execution_time_expression(timestamp):
    # execution takes 60 +- 30 seconds
```

```
    return 60+random.randint(-30,+30)

q = Queue()
s = Source(output=q,
           creation_tax_expression=creation_tax_expression,
           execution_time_expression=execution_time_expression)
p = Process(inputs=[q])

simul = Simulator(components=[q,s,p], verbose=True)

steps_number = 60*60 #1 second each timestamp, run for 60 minut
simul.run(untill=steps_number)

q_size = len(simul.components['queue'][0])
print q_size
```

Cada objeto recebe um determinado número de parâmetros, como a taxa de criação de eventos para o componente gerador, e a taxa de consumo de eventos, no caso do component `p`.

Em seguida, é criada uma instância do simulador (`simul = Simulator(...)`) passando como argumentos para esse os componentes envolvidos na simulação.

O método que dispara a simulação é o método `run` da classe `Simulator`. Este método recebe como parâmetro a quantidade de iterações (*steps*) que a simulação deverá executar. Se for do interesse que a simulação ocorra por um tempo indeterminado até o seu cancelamento manual, basta passar o valor zero ou `None` como parâmetro.

5.3 Variação de um Modelo para Comparação

Neste exemplo, é simulado o modelo ilustrado pelas figuras 5.1 e 5.2. O modelo simula uma situação onde existem duas filas paralelas com frequências distintas de inserção de eventos. A frequência em que um novo evento é adicionado à fila de eventos obedece às funções `ct_1` e `ct_2`, e o tempo que cada evento leva para ser processado é expresso pela função `execution_time_expression`.

A primeira parte do *script* simula o modelo ilustrado pela figura 5.1, onde tem-se apenas um consumidor para as duas filas. Já na segunda parte do mesmo *script*, é simulada a mesma situação, porém com dois consumidores, conforme ilustrado na figura 5.2.

Listing 5.4: Exemplo de um sistema mais complexo

```
from t100.core.simulator import Simulator
from t100.core.base_components import *

def execution_time_expression(timestamp): return 60 + random.randint(-30

def ct_1(timestamp): return 1.0 / (60 * 5)

def ct_2(timestamp): return 5.0 / (60 * 5)

steps = 60*60 # por quantos steps a simula   o deve executar

#INICIO DA SIMULACAO COM UM UNICO CONSUMIDOR DE EVENTOS
acc1 = acc2 = 0
q1 = Queue(); q2 = Queue()
s1 = Source(output=q1,
            creation_tax_expression=ct_1,
            execution_time_expression=execution_time_expression)
```

```
s2 = Source(output=q2,
            creation_tax_expression=ct_2,
            execution_time_expression=execution_time_expression)
p = Process(inputs=[q1,q2])
#Adicionando componentes a simulacao
simul = Simulator(components=[q1,q2,s1,s2,p], )
#executando a simulacao
simul.run(untill=steps)
q1_size = len(simul.components['queue'][0])
q2_size = len(simul.components['queue'][1])
#imprimindo o resultado (tamanho da fila)
print q1_size, q2_size
#FIM DA SIMULACAO DO MODELO COM UM UNICO CONSUMIDOR DE EVENTOS
#INICIO DA SIMULACAO COM DOIS CONSUMIDORES DE EVENTOS
acc1 = acc2 = 0
q1 = Queue(); q2 = Queue()
s1 = Source(output=q1,
            creation_tax_expression=ct_1,
            execution_time_expression=execution_time_expression)
s2 = Source(output=q2,
            creation_tax_expression=ct_2,
            execution_time_expression=execution_time_expression)
p1 = Process(inputs=[q1,q2])
p2 = Process(inputs=[q1,q2])
simul = Simulator(components=[q1,q2,s1,s2,p1,p2])
simul.run(until=steps)
q1_size = len(simul.components['queue'][0])
q2_size = len(simul.components['queue'][1])
print q1_size, q2_size
#FIM DA SIMULACAO DO MODELO COM DOIS CONSUMIDORES DE EVENTOS
```

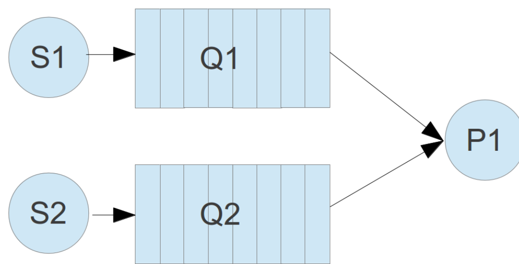

Figura 5.1: Modelo com um consumidor

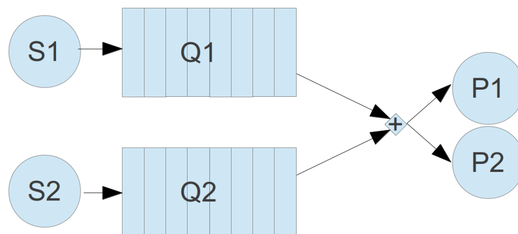

Figura 5.2: Modelo com dois consumidores

Como resultado, a simulação devolve o tamanho final de cada fila (`q1` e `q2`). Pode, com isso, avaliar o impacto de se adicionar um processo consumidor no sistema.

5.4 Exemplo de Simulação Distribuída

O exemplo da listagem 5.5 ilustra como o mesmo código descrito pela seção 5.2 pode ser executado em um sistema distribuído.

Para que o código descrito pela listagem 5.3 execute em um sistema distribuído, basta informar no código quais unidades na rede estão aptas a receber processos lógicos. Isto pode ser feito através de um arquivo que mapeie os nós da rede que estão preparados para receber os processos lógicos (listagem 5.6).

Em seguida, cada unidade de rede listada deve executar uma instanciando de

`Environment`. O *framework* usa o *middleware* de comunicação para distribuir os processos descritos pelo modelo por entre os nós disponíveis na rede.

Para que o exemplo listado funcione, é necessário que todos os nós citados no arquivo de configuração possuam instâncias válidas do ambiente de simulação. Estes ambientes (*environment*) devem estar executando no endereço (IP e porta) descritos pela configuração.

Listing 5.5: Simulação distribuída

```
from t100.components.components import *
from t100.simtool.distributed_env import Environment

import random
import time
import sys

def creation_expression(timestamp):
    return 1.0/1000

def execution_expression(timestamp):
    return 10 + random.randint(-5,+5)

ip,port = '192.168.0.11',11111

process = QueuedProcess()
source = Source(output=process,
                creation_tax_expression=creation_expression,
                execution_time_expression=execution_expression,)

cfg = open('distributed_01.json').read()
env = Environment(ip, port, cfg, verbose=True)
```

```
env.populate([source, process])

env.start_simulation(untill=1000)
```

Listing 5.6: "Arquivo de configuração dos nós do sistema"

```
{
"number_of_nodes":2,
"nodes":[
    "192.168.0.11:11111",
    "192.168.0.12:11111"
    ]
}
```

Uma vez que essas condições são satisfeitas, os componentes descritos pela modelagem do sistema são distribuídos pelos nós do sistema e a execução é iniciada.

5.5 Considerações Finais

Este capítulo trouxe alguns exemplos de como um usuário do *framework* de simulação utilizaria o *framework* aqui proposto para descrever um modelo a ser simulado e executar a simulação.

Os mecanismos de comunicação (oferecido pelo *middleware* de comunicação) e de distribuíção (oferecidos pelo *framework* de simulação) encapsulam decisões internas, deixando o seu funcionamento transparente para o usuário.

6 Discussões Finais e Conclusões

Este trabalho teve como meta desenvolver uma arquitetura de *middleware* de comunicação para simplificar a comunicação e a migração de processos lógicos para a implementação de um *framework* de simulação distribuída de eventos discretos.

Ao se criar esta camada, que se responsabiliza por todo o processo de tratamento de comunicação e migração de processos no sistema, caberia ao desenvolvedor do *framework* apenas a manipulação destes elementos de maneira isolada. Este desenvolvedor não precisaria se envolver com minúcias referentes à manipulação de endereços de processos lógicos, retransmissão de mensagens transientes, migração de processos, entre outros.

Este trabalho apresentou uma proposta de arquitetura para esta camada de comunicação e trouxe uma implementação para demonstrar o funcionamento deste *middleware*, provendo as ferramentas de migração e comunicação entre os processos lógicos. Esta implementação foi realizada utilizando a linguagem Python, uma linguagem de alto nível orientada à objetos e que possibilita uma rápida prototipação, além de ser amplamente utilizada em computação científica.

Por fim, salienta-se que os testes realizados na implementação do *middleware* apresentada neste trabalho indicam que esta cumpre o seu propósito principal de redirecionamento de mensagens e migração de processos lógicos no sistema distribuído.

6.1 Contribuições deste Trabalho

A principal contribuição deste trabalho é a proposta de uma camada de comunicação que permite a troca de mensagens entre processos de maneira transparente e independente da sua localização no ambiente de simulação distribuída. Outra contribuição, é a proposta de desenvolvimento de um mecanismo de migração de processos lógicos utilizando esta camada de comunicação como suporte. Com isso, obtém-se uma comunicação homogênea entre processos lógicos, independente da sua localização no sistema de simulação.

Uma implementação inicial desta camada de comunicação foi desenvolvida utilizando a linguagem Python. Esta implementação demonstrou que a criação de ambientes virtuais (aqui denominados *environment*) com o intuito de isolar a comunicação dos processos lógicos com o mundo exterior, interceptando suas mensagens, provê um mecanismo para redirecionamento de mensagens que mantém a comunicação de maneira homogênea, mesmo durante a migração de um processo lógico.

Por fim, este trabalho sugere uma alternativa aos agentes móveis para a implementação de mobilidade de processos lógicos em um sistema de simulação distribuída.

6.2 Sugestões para Trabalhos Futuros

Este *middleware* é uma ferramenta de apoio aos profissionais que desejam implementar um programa de simulação distribuída de eventos discretos. Desta forma, a partir dele, vários trabalhos podem ser desenvolvidos, tanto na área de *frameworks* para aplicaões distribuídas quanto na área de mobilidade e comunicação de processos.

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

- Otimização do mecanismo de envio de mensagens, concatenando mensagens a ser entregues a um mesmo ambiente a fim de serem enviadas em um único pacote;
- Avaliação de diferentes bibliotecas de troca de mensagens em substituição à implementação padrão de *sockets* provida pela linguagem Python;
- Implementação de um protocolo de sincronização (*Time Warp* ou *Rollback* Solidário) utilizando a arquitetura de comunicação aqui discutida;
- Implementação de um mecanismo de balanceamento de cargas utilizando a arquitetura de comunicação e migração aqui discutida;
- Levantamento do custo do redirecionamento de mensagens transientes, recebidas durante uma migração;
- Implementação da mesma arquitetura proposta em diferentes linguagens (como C, Java, Go, etc.) a fim de se realizar um *benchmark* entre as implementações;
- Execução de simulações utilizando modelos reais, a fim de se verificar o desempenho do *middleware* em situações de alta demanda;
- Implementação de diferentes métricas de escalonamento, para avaliação do impacto destes utilizando o *middleware* de comunicação.

Referências Bibliográficas

BAEZNER, D.; LOMOW, G.; UNGER, B. W. *A parallel simulation environment based on time warp*. [S.l.]: International Journal in Computer Simulation, 1994. 183–207 p.

BANKS, J. et al. Discrete-event system simulation. In: _____. [S.l.]: Prentice Hall, 2010. p. 3.

BHAM, G. H.; BENEKOHAL, R. F. A high fidelity traffic simulation model based on cellular automata and car-following concepts. *Transportation Research Part C: Emerging Technologies*, v. 12, n. 1, p. 1 – 32, 2004. ISSN 0968-090X. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0968090X03000573>.

BRANCO, K. R. L. J. C. *Índices de carga e desempenho em ambientes paralelos/distribuídos - modelagem e métricas.* Tese (Doutorado) — USP - São Carlos, SP, 2004.

CHAN, W. K. V.; SON, Y.-J. Simulation of emergent behavior and differences between agent-based simulation and discrete-event simulation. Winter Simulation Conference, 2010.

CICIRELLI, F.; NIGRO, L. An agent framework for high performance simulations over multi-core clusters. In: . [S.l.]: 13th International Conference on Systems Simulation, Singapore, 2013. p. 49–60.

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

DAS, S. Gtw: A time warp system for shared memory multiprocessors. In: *Proceedings of the 26th Conference on Winter Simulation*. [S.l.: s.n.], 1994. p. p. 1332–1339.

FUJIMOTO, R. M. *Parallel and Distribution Simulation Systems*. New York, NY, USA: John Wiley & Sons, Inc., 1999. ISBN 0471183830.

HIGHAM, D. J. An algorithmic introduction to numerical simulation of stochastic differential equations. *Society for Industrial and Applied Mathematics*, v. 43, n. 3, p. 525–546, aug. 1971.

JEFFERSON, D. R. Virtual time. *ACM Trans. Program. Lang. Syst.*, ACM, New York, NY, USA, v. 7, n. 3, p. 404–425, jul. 1985. ISSN 0164-0925. Disponível em: <http://doi.acm.org/10.1145/3916.3988>.

KUHLMAN, C. J. et al. A general-purpose graph dynamical system modeling framework. Winter Simulation Conference, 2011.

LANGE, D. B.; OSHIMA, M. *Seven Good Reasons for Mobile Agents*. 1999. Communications of the ACM.

LYNCH, P. The origins of computer weather prediction and climate modeling. *Journal of Computational Physics*, v. 2, p. 3431–3444, 2007.

MADIREDDY, M.; KUMARA, D. J. M. S. An agent based model for evacuation traffic management. Winter Simulation Conference, 2011.

MARTIN, D. E.; MCBRAYER, T. J.; WILSEY, P. A. Warped: A time warp simulation kernel for analysis and application development. In: . [S.l.]: Proceedings of the 29th Hawaii International Conference on System Sciences, 1996. v. 1, p. 383–386.

MATLOFF, N. *A Discrete-Event Simulation Course Based on the SimPy Language*. october 2011. Disponível em: <http://heather.cs.ucdavis.edu/ matloff/simcourse.html>.

MCQUILLAN, J. M.; WALDEN, D. C. Some considerations for a high performance message-based interprocess communication system. In: . [S.l.]: Proceedings of the 1975 ACM SIGCOMM/SIGOPS workshop on Interprocess communications, 1975. v. 9.

MIDDLETON, V. Simulating small unit military operations with agent-based models of complex adaptive systems. Winter Simulation Conference, 2010.

MILOJICIC, D. S. et al. Process migration. *ACM Computing Surveys*, v. 32, p. 2000, 2000.

MISRA, J.; CHANDY, K. M. Distributed simulation: A case study in design and verification of distributed programs. In: ______. [S.l.]: IEEE Transactions on Software Engineering, 1979. v. 1, n. 5, p. 440–452.

MOREIRA, E. M. *Rollback Solidário: Um novo Protoloco Otimista para Simulação Distribuída*. Tese (Doutorado) — USP - São Carlos, SP, 2003.

NAGEL, L.; ROHRER, R. Computer analysis of nonlinear circuits, excluding radiation (cancer). *Solid-State Circuits, IEEE Journal of*, v. 6, n. 4, p. 166 –182, aug. 1971. ISSN 0018-9200.

OSKOOI, A. F. et al. MEEP: A flexible free-software package for electromagnetic simulations by the FDTD method. *Computer Physics Communications*, v. 181, p. 687–702, January 2010.

PARK, H.; FISHWICK, P. A. A gpu-based application framework supporting fast discrete-event simulation. In: . [S.l.]: Transactions of the society for modeling and simulation interntional, 2010. v. 86, p. 613–628.

PERRONE, L. F. et al. Sassy: A design for a scalable agent-based simulation system using a distributed discrete event infrastructure. In: *Proceedings of the 2006 Winter Simulation Conference*. [S.l.: s.n.], 2006.

RAILSBACK, S. F.; LYTINEN, S. L.; JACKSON, S. K. Agent-based simulation platforms: Review and development recommendations. In: . [S.l.]: Transactions of the society for modeling and simulation interntional, 2012. v. 82, p. 609–623.

REYNOLDS, P. F. et al. Making parallel simulations go fast. In: *Proceedings of the 1992 Winter Simulation Conference*. [S.l.]: Press, 1992. p. 646–656.

RIBEIRO, A.; WALBON, G.; TAKAHASHI, W. *Agentes móveis e Simulação Distribuída*. 2009.

RIEHLE, D. *Framework Design: A Role Modeling Approach*. Tese (Doutorado) — ETH Zürich, 2000.

SIVANANDAM, S. N.; VISALAKSHI, P. Dynamic task scheduling with load balancing using parallel orthogonal particle swarm optimisation. *Int. J. Bio-Inspired Comput.*, Inderscience Publishers, Inderscience Publishers, Geneva, SWITZERLAND, v. 1, n. 4, p. 276–286, abr. 2009. ISSN 1758-0366. Disponível em: <http://dx.doi.org/10.1504/IJBIC.2009.024726>.

SOKOLOWSKI, J. A.; BANKS, C. M. Modeling and simulation: Real-world examples. In: _____. [S.l.]: Wiley, 2008. p. 6.

SRINIVASAN, S.; REYNOLDS JR., P. F. Npsi adaptive synchronization algorithms for pdes. In: *Proceedings of the 27th conference on Winter simulation*. Washington, DC, USA: IEEE Computer Society, 1995. (WSC '95), p. 658–665. ISBN 0-7803-3018-8. Disponível em: <http://dx.doi.org/10.1145/224401.224705>.

STEINMAN, J. S. Speedes: A multiple synchronization environment for parallel discrete event simulation. *International Journal in Computer Simulation*, v. 2, p. 251–286, 1992.

TAI, H.; KOSAKA, K. The aglets project. *Commun. ACM*, ACM, New York, NY, USA, v. 42, n. 3, p. 100–101, mar. 1999. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/295685.295882>.

TAVAKOLI, S.; MOUSAVI, A.; KOMASHIE, A. A generic framework for real-time discrete event simulation (des) modelling. In: . [S.l.]: Simulation Conference, 2008. WSC 2008. Winter, 2008. p. 1931–1938.

VOORSLUYS, B. L. *Influência de políticas de escalonamento no desempenho de simulações distribuídas.* 2006.

ZHANG, H.; TAN, H. B. K.; MARCHESI, M. The distribution of program sizes and its implications: An eclipse case study. In: . [S.l.: s.n.], 2009.