Antonio Ribeiro Alves Júnior

Arquitetura de um *middleware* de comunicação para implementação de Simulação Distribuída

Itajubá - MG

01 de Outubro de 2012

Antonio Ribeiro Alves Júnior

Arquitetura de um *middleware* de comunicação 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 Ciência e Tecnologia da Computação.

Orientador:
Prof. Dr. Edmilson Marmo Moreira

Universidade Federal de Itajubá - Unifei
Instituto de Engenharia de Sistemas e Tecnologias da Informação
Engenharia da Computação

Itajubá - MG

01 de Outubro de 2012

Sumário

Lista de Figuras

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 por estar inacessível, ou por ser considerado perigoso ou inaceitável comprometer este sistema. Também pode ser empregada quando deseja-se 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 (**??**), 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, auxílio na tomada de decisão sem a necessidade de intervir no sistema real, facilidade na manipulação e alteração dos modelos, economia de recursos (físicos e financeiros) durante a tomada de decisões, dentre outros.

Para utilizar a simulação é necessário construir e analisar modelos que representam 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 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.

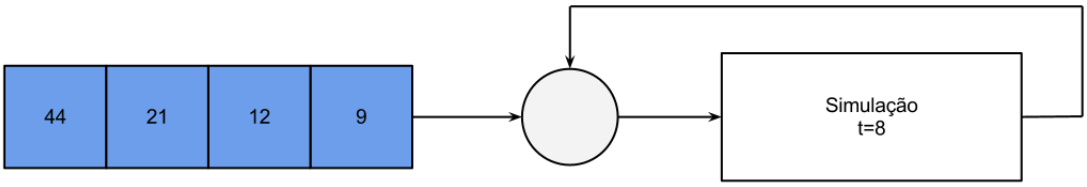

Figura 1.1: Simulação Sequencial.

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.

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

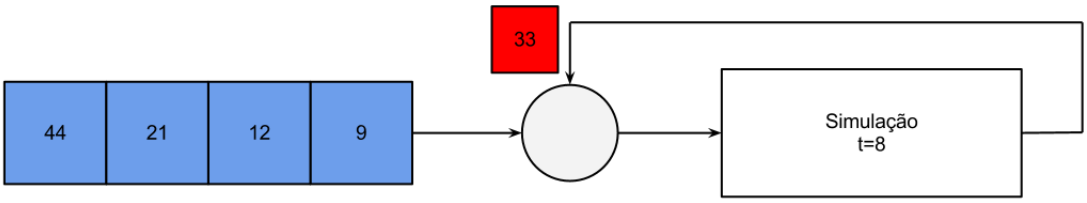

Figura 1.2: Geração de um novo evento.

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 encontradas para se solucionar estes problemas foi dividir o tratamento dos diversos eventos entre vários processadores de uma mesma máquina paralela ou sobre 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 de observação 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 sistemas, 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 sistemas existem algorítmos capazes de analisar o sistema, considerando fatores como volume de troca de mensagem e necessidade de processamento de cada processo lógico e mapear cada processo para um determinado nó do sistema.

Porém, para que isso seja possível, o processo lógico deve ser capaz de efetuar uma migraçõa, 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 que ela foi interrompida.

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

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 dois protocolos utilizados para sincronização de simulação distribuída.

O terceiro capítulo aprofunda-se na proposta do projeto, apresentando a arquitetura básica do *middleware* de comunicação que visa proporcionar mobilidade aos processos lógicos no sistema de simulação distribuída.

Os capítulos quatro e cinco apresentam, consecutivamente, a arquitetura interna do *middleware* de comunicação e a arquitetura de um micro-*framework* desenvolvido para demonstrar o funcionamento do *middleware* de comunicação.

Por fim o capítulo seis traz alguns detalhes de implementação do *middleware* proposto que o autor julgou conveniente descrever, seguido pela conclusão, compondo o sétimo capítulo.

2 Simulação Distribuída de Eventos Discretos

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, auxílio na tomada de decisão sem a necessidade de intervir no sistema real, facilidade na manipulação e alteração dos modelos, economia de recursos (físicos e financeiros) durante a tomada de decisões, dentre outros.

Por demandar um intenso processamento computacional, a simulação por vezes pode se tornar uma opção demasiadamente custosa. Melhores algorítmos e métodos de simulação mais modernos são maneiras de tornar a simulação mais eficiente. Em paralelo à 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 criadas diversos novos requisitos ao sistema. Ítens como sincronização dos processos, comunicação entre os processos, balanceamento de cargas no sistema, entre outros devem ser gerenciados pela aplicação de simulação a fim de se garantir um resultado coerente do processo de simulação.

2.1 Princípios da Simulação Baseada em Eventos Discretos

No paradigma da simulação discreta, três variáveis 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

Cada evento a ser executado possui uma marca de tempo (*timestamp*) que determina quando, no tempo lógico 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 processa-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 (**??**). Para isto, o sistema passou a ser dividido em processos lógicos que representam um processo de sistema real. Uma simulação é um conjunto com n processos lógicos $p_1, p_2, p_3, ..., p_n$.

Assim como o sistema de simulação possui um relógio lógico que representa o tempo lógico 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 de simulação para aquele processo lógico. Isso garante que os eventos a serem executados por um determinado processo lógico sejam executados no tempo lógico 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.

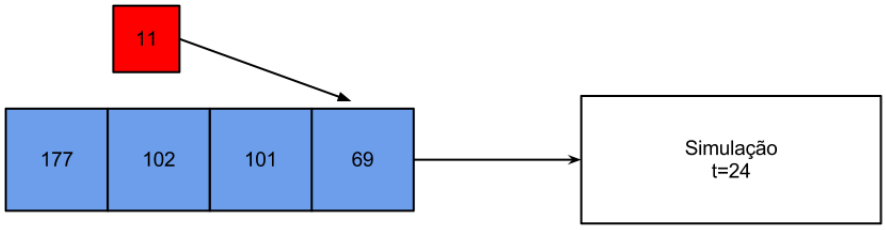

Figura 2.1: Mensagem *straggler*.

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

Assim como todo evento discreto, um evento enviado de um processo lógico à 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 lógico 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 recebeu a mensagem. Assim como ilustrado na figura 2.1, o processo em questão possui seu relógio lógico no tempo $t = 24$ e recebe uma mensagem para ser processada no tempo $t = 11$. Como o evento a ser processado possui um *timestamp* menor que o relógio interno do processo lógico, o sistema encontra-se em um momento de inconsistência. Um sistema de simulação distribuída deve ser capaz de contornar tal situação.

2.2 Categorias de protocolos de simulação

Para se 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 ao seu comportamento perante um inconsistência de causa e efeito no sistema: protocolos conservadores e protocolos otimistas.

Um protocolo conservador 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 acontecem.

2.2.1 Protocolos conservaticos

Parágrafo 1: citar Srinivasan and Reynolds, explicando o conceito básico de um protocolo conservativo

Parágrafo 2: citar as implementacoes de [chandy e misra] e bryant, citando a necesidade de canais estaticos

Parágrafo 3: citar problema de bloqueio por esperar mensagem que contém um lvt inferior, citar também como contornar esse deadlock

Parágrafo 4: explicar porque é difícil resolver o problema de deadlock

Parágrafo 5: citar a desvantagem do conservativo por não aproveitar todo o paralelismo

Parágrafo 6: conclusão pessoal de porque não utilizar o protocolos conservativos

2.2.2 Protocolos Otimistas

Parágrafo 1: explicação inicial do conceito de protocolo otimista e citar o conceito de rollback

Parágrafo 2: Citar o time warp, contextualizando como foi desenvolvido, em qual época e por quem

Parágrafo 4: Citar o Rollback Solidário como alternativa ao Time Warp

2.3 O protocolo Time Warp

Parágrafo 1: Explica por que o Time Warp é otimista, citando o fato de nao impedir a ocorrência de erros de causa e efeito

Parágrafo 2: Citar como um erro de causa e efeito pode acontecer, e o que deve ser feito quando isso acontece: o rollback

Parágrafo 4: Citar implementações do protocolo

Parágrafo 5: Explicar o comportamento individual de um processo lógico (tirar o eventos com menor timestamp da lista de eventos futuros, etc.)

2.3.1 Detecção e Tratamento de Inconsistências

Parágrafo 1: Mostrar que o sistema pode receber mensagens que causam erro de causa e efeito.

Parágrafo 2: Explicar o comportamento ao receber uma mensagem straggler.

Parágrafo 3: definir a diferença entre rollback primário e rollback secundário

2.3.2 Anti-Mensagens

Parágrafo 1: Explicar o que é uma antimensagem

Parágrafo 2: mostrar que antimensagens podem se referir a um evento já processado, ou a um evento que ainda esta na fila de eventos futuros

Parágrafo 3: Tratar o caso da antimensagem chegar antes da mensagem

2.3.3 Considerações Finais

2.4 O protocolo Rollback Solidário

Parágrafo 1: Apresentar a principal diferença do rollback solidário

Parágrafo 2: explicar que, em caso de erro de causa e efeito, todos os processos executam rollback em conjunto

Parágrafo 3: introduzir o conceito de checkpoint, checkpoint global e checkpoint local.

2.4.1 Comportamento Geral do Protocolo Rollback solidário

Trazer uma idéia básica do Rollback solidário em linhas gerais

2.4.2 Estados Locais e Estados Globais

Parágrafo 1: Definição de um estado local como sendo os valores das variáveis do processo em questão em um determinado tempo

Parágrafo 2: Definição de estado global como sendo um conjunto de estdaos de cada processo.

2.4.3 Cortes Globais Consistentes

Parágrafo 1: Apresentar o conceito de precedência causal

Parágrafo 2: Definir corte

Parágrafo 3: definir corte consistente utilizando precedência causal

2.4.4 Relógios Lógicos e Relógios Vetoriais

Parágrafo 1: Apresentar o conceito de relógio lógico

Parágrafo 2: Mostrar que o relógio lógico não representa a precedência causal em um sistema distribuído

Parágrafo 3: Apresentar o conceito de relógio vetorial

2.4.5 Checkpoints Globais Consistentes

Parágrafo 1: Definir o que é um checkpoint: um ponto de retorno que foi, de alguma forma, armazenado de forma persistente

Parágrafo 2: a necessidade de se garantir que um checkpoint seja consistente

Parágrafo 3: associação de checkpoint consistente com corte consistente

2.4.6 Obtenção de Checkpoint Semi-Síncrono

Parágrafo: introduzir o processo observador

Parágrafo: Introduzir o vetor de dependência

Parágrafo: Mostrar como a troca de mensagem deve levar o vetor de dependências, e como este deve ser atualizado no processo que recebeu a mensagem

Parágrafo: mostrar como o processo observador monta uma linha de recuperação através da matriz de dependências

Parágrafo: demonstrar que um checkpoint global consistente é formado por checkpoints locais que não possuem relação causal entre si

2.4.7 Tratamento dos Rollbacks na Abordagem Semi-Síncrona

Parágrafo: Explicar o comportamento de um processo ao receber uma mensagem straggler

Parágrafo: Falar sobre a atuação do processo observador (escolher a linha de retorno, avisar o rollback à todos em broadcast, etc.)

Parágrafo: Falar sobre o reinício da simulação após o Rollback

2.4.8 Considerações Finais

2.5 Balanceamento de cargas

Parágrafo: apresentar o balanceamento de carga como fator determinante no desempenho de uma simulação distribuída

Parágrafo: Introduzir o conceito de escalonamento de processos como meio de prover o balanceamento de cargas

Parágrafo: por fim mostrar que para prover o balanceamento de cargas temos que possibilitar que um processo lógico migre de um nó do sistema para outro

2.5.1 O Uso de Agentes Móveis Para Prover Mobilidade

Parágrafo: O que é um agente móvel

Parágrafo: Falar sobre a implementação feita por Antonio, Walbon e Takahashi - 2010

Parágrafo: Outras implementações feitas usando agentes móveis

2.5.2 Requisitos Para Mobilidade de um Processo Lógico

Parágrafo: explicar quais as funcionalidades de agentes móveis que precisamos (migração)

Parágrafo: Abstrair o conceito de ambiente e processo lógico: um ambiente abriga diversos processos lógicos

Parágrafo: Mostrar a idéia de migrar um processo de um ambiente para outro, a fim de prover o balanceamento

2.5.3 Considerações Finais

3 Proposta deste projeto

Este capítulo traz uma breve descrição das características intrínsecas a um *framework*, seguida por uma definição das principais características que determinam a sua natureza, consideradas determinantes para o desenvolvimento deste trabalho. em seguida é apresentada algumas soluções existentes para o desenvolvimento de simulações distribuídas de eventos discretos e por fim é apresentado o modelo desenvolvido neste trabalho, tanto como suas principais características e diferenciações em relação às soluções atualmente existentes.

3.1 Um framework para simulação distribuída

Escrever uma aplicação de simulação de eventos discretos distribuída é uma tarefa de grandes proporções. Todo o tratamento de sincronização, comunicação, troca de mensagens, manipulação de objetos remotos, entre outras tarefas, acarretam no surgimento de diversas 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 cuidado com o desempenho (entram neste ítem balanceamento de carga, *design* eficiente dos algorítmos utilizados, etc) e permissividade ao erro (a probabilidade de se intruduzir um erro em um código aumente proporcionalmente ao tamanho deste código, (ZHANG; TAN; MARCHESI, 2009)) e obtem-se neste ponto 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 em 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. Para prover tal separação entre a aplicação escrita pelo usuário e o *framework*, este compromete-se a prover três funcionalidades essenciais: encapsulamento, transparência e reusabilidade.

3.1.1 Encapsulamento

Uma das funções de um *framework* 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 elementos lógicos para representar o modelo a ser simulado. Quando encapsulamos o funcionamento de uma fila de eventos ou o de um processo lógico em uma classe, por exemplo, estamos isolando sua implementação e seus detalhes do usuário final do *framework*. Ao usuário cabe reutilizar estes componentes e, quando julgar necessário, criar componentes baseados nesses primitivos.

Outros exemplos de ocasiões em que se pode aplicar o encapsulamento é tanto nos algorítmos responsáveis pelo balanceamento de cargas no sisteam quanto nos mecanismos de sincronização de processos lógicos. A possibilidade de encapsular esses componentes do *framework* em classes separadas nos possibilita o intercâmbio de diferentes implementações que solucionam um mesmo problema. Com uma interface bem desenvolvida, pode se criar, por exemplo, encapsulamentos distintos para o protocolo *Time Warp* e para o protocolo *Rollback* Solidário. Isso permitiria

que o usuário escolhesse, antes de iniciar a simulação, qual protocolo pretende adotar no processo.

3.1.2 Transparência

A proposta de se escrever um código que seja ao mesmo tempo fácil de se implementar pelo usuário do *framework* e eficiente em sua execução projeta-se diretamente na utilização de diversas camadas que ao mesmo tempo esconde do usuário do *framework* 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: sincronização, comunicação, migração e balanceamento de carga em um sistema distribuído de simulação de eventos discretos), deixando ao usuário apenas a função de desenvolver o modelo, sem a necessidade de se preocupar com questões que estão fora de seu domínio.

O conceito de transparência neste caso remete-se na intenção do *framework* que é deixar invisível ao seu usuário toda e qualquer decisão que não compete à construção do seu modelo a ser simulado. Isso acaba trazendo para a construção do *framework* algumas responsabilidades quanto a tomadas de decisões sobre *design* de software, implementação de algorítmos considerados decisivos, entre outras.

3.1.3 Reusabilidade

Ao se eleborar um *framework*, o responsável pelo seu *design* deve permitir que componentes internos deste sejam trocados ou mesmo customizados. Isso garante que que o mesmo código escrito para ser executando em um determinado *framework* continue a funcionar mesmo depois da troca de algum componente interno deste *framework*.

No caso da simulação distribuída, componentes como os protocolos de sincronização ou o sistema de balanceamento de cargas poderiam ser plugáveis. Isto permitiria customizar o comportamento do *framework* apenas selecionando estes componentes, o que permitiria que simulássemos o mesmo modelo (sem alterações) usando diferente soluções para sincronização, balanceamento de carga, etc. Esta flexibilidade garante a reutilização de código, economizando no desenvolvimento da simulação e dinamizando a comparação entre diferentes soluções para um mesmo modelo.

3.2 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 (ALVES; WALBON; 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* provém soluções para a a troca de mensagem entre diferentes processos. Essas soluções provém 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 preparados para o redirecionamento de mensagens enviadas à processos que migraram para um nó diferente do seu nó de origem.

Outras soluções como (PERRONE et al., 2006) e (), assim como (ALVES; WALBON; TAKAHASHI, 2009), baseiam-se nos agentes móveis para se desenvolver a aplicação de simulação distribuída. As soluções baseadas em agentes móveis

proporcionama uma mobilidade aos processos lógicos, útil ao balanceamento de carga. Porém, assim como as bibliotecas de comunicação citadas anteriormente, não estão preparadas para redirecionar as mensagens destinadas à processos que migraram de seus nós de origem.

Visando isso este trabalho propõe apresentar uma solução, em formato de *fremowork*, para o desenvolvimento de simulações distribuídas de eventos discretos. Este *framework* deve possibilitar ao seu usuário descrever o modelo a ser simulado de forma simples, e simulá-lo em diferentes configurações (diferentes protocolos de sincronizaçao, de forma centralizada ou distribuída, etc) apenas modificando suas configurações, sem a necessidade de alterar o código que descreve o modelo a ser simulado.

Para que sejam sustentados tanto os mecanismos de sincronização da simulação quanto o balanceamento de cargas, é vital que o *framework* supra, internamente, as necessidades básicas de comunicação, troca de mensagens, migração de processos lógicos e comunicação grupal entre os processos. Estas funcionalidades são providas pelo *middleware* de comunicação do *framework*.

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

3.3 Arquitetura proposta

A figura 3.1 apresenta uma visão ampla da arquitetura do framework de simulação distribuída aqui proposto.

A camada superior, denominada aplicação, é a interface pela qual o usuário

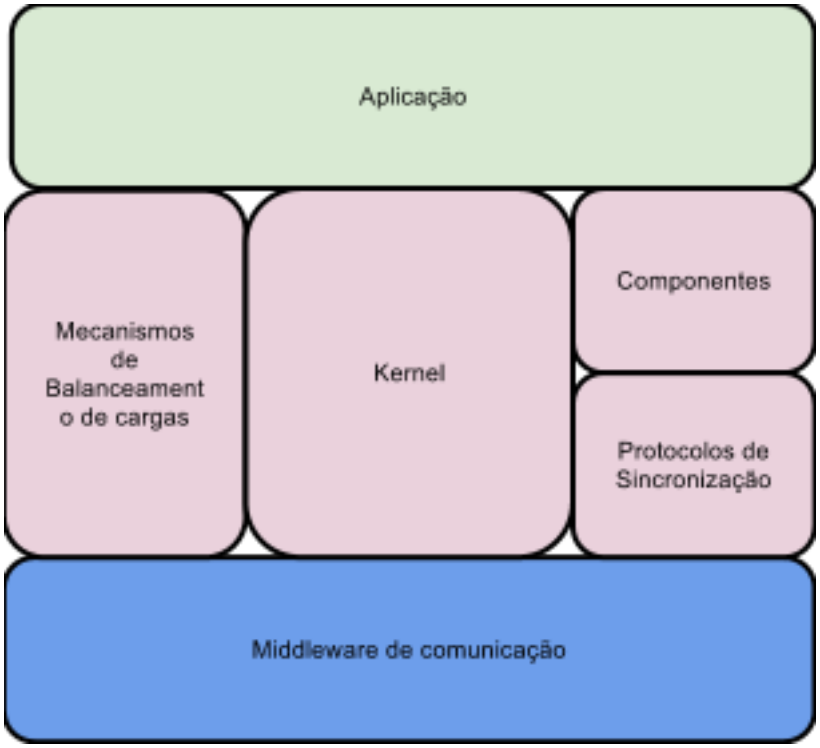

Figura 3.1: Camadas da arquitetura do *framework*.

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 algorítmos 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. 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.

Conforme descrito na seção 3.2, os mecanismos de troca de mensagens existentes não possuem 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 horigem para um novo ambiente tenhamos que, de maneira explícitar, avisar 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.

Nos casos que mais se aproximam deste requesito do sistema, os agentes móveis, há uma transparência na troca de mensagens até o momento em que um agente migra para um diferente nó no sistema. Ao se mover para um ambiente diferente do seu ambiente de origem, implementações de agentes móveis como *Aglets* requerem uma intervenção do usuário para que se refatore os valores dos endereços físicos dos objetos para que se possa continuar a somunicação. O middleware aqui proposto visa resolver este problema, tornando a comunicação entre processos completamente transparente para o seu usuário, mesmo após a migração de uma componente de seu nó de origem.

3.4 Organização deste documento

Os capítulos seguintes tratam da descrição detalhada da arquitetura do projeto e de sua implementação. O capítulo quatro trata da arquitetura do *middleware* de comunicação. O capítulo cinco traz detalhes da arquitetura do *framework* de simulação.

No capítulo seis é demonstrado detalhes de implementação que o autor julga conveniente descrever neste documento e alguns exemplos de simulação utilizando o framework desenvolvido.

Por fim o capítulo sete traz as discussões sobre os resultados desse trabalho, tanto quanto conclusões e propostas para sua continuaidade em trabalhos futuros.

4 Arquitetura do middleware de comunicação

Conforme mencionado no capítulo anterior, todo o desenvolvimento do *framework* proposto neste trabalho se faz em cima de uma camada denominada *middleware* de comunicação. Este capítulo descreve em detalhes a arquitetura do proposto *middleware* de comunicação, tais como suas partes e suas principais funções: troca de mensagens e migração de processos lógicos.

4.1 Módulos do middleware

A arquitetura do *middleware* é dividida em dois módulos principais: o ambiente (*envinonment*) e o processo lógico (*process*). 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*.

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 por 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 *process* são abstrações lógicas que representam a simulação baseado em eventos discretos. Fisicamente tanto o ambiente quanto o processo lógico são instâncias de objetos que devem ser implementadas extendendo 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 conteúde 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 ítem 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; WALDEN, 1975) sobre os canais de comunicação inter-processos:

- 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 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 proporcionar a capacidade de balanceamento de cargas em uma simulação distribuída. O mecanismo de migração, descrito na Seção 4.4, 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.

4.2 O módulo 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 desejamos manipular simultaneamente características de diversos processos que possuêm em comum o fato de estarem no mesmo ambiente físico (como por exemplo migrar ou serializar todos os processos). Mas principalmente se justifica quando deseja-se ter uma camada que seja responsável por gerenciar o ciclo de vida destes processos, como a criação, migração e destruição destes processos lógicos.

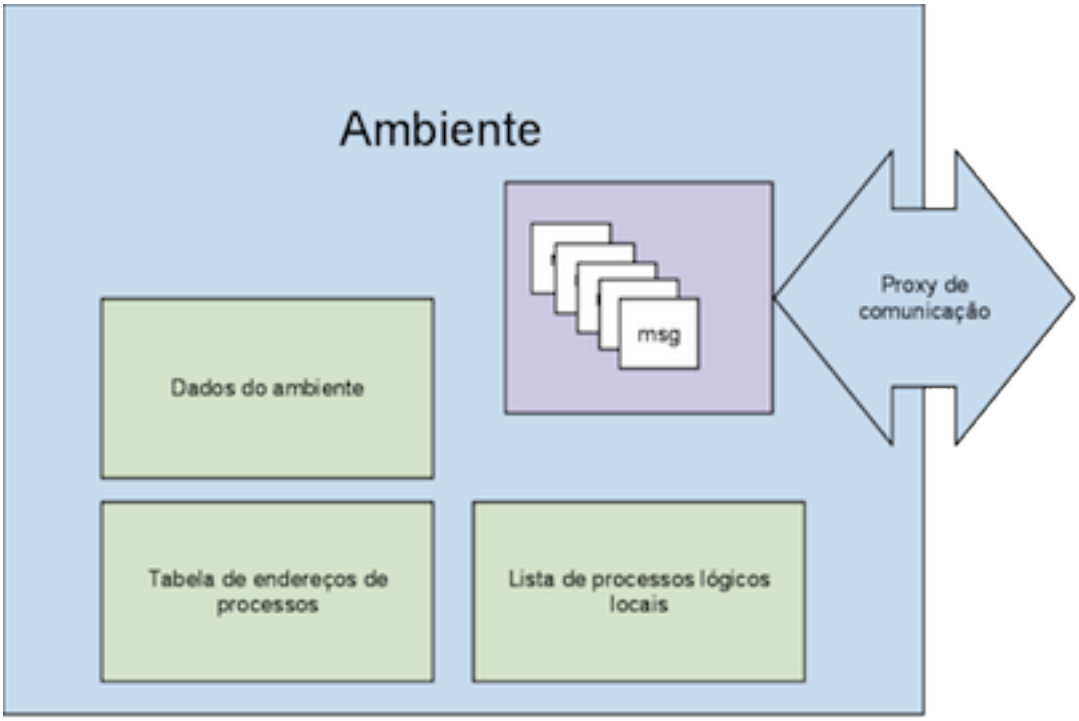

Figura 4.1: A arquitetura interna de um *environment*.

4.2.1 Estrutura interna

Internamente, o *environment* apresenta as seguintes estruturas básicas (conforme ilustrada na Figura 4.1):

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

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 toda o ciclo de vida da simulação, denominado *Unique Environment Identifier - UEI*. Este endereço é um número natural, e o representa no sistema de simulação.

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

4.2.2 *Proxy*

O *proxy* é a camada do *environment* que é responsável por toda troca de mensagem 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 4.5

O *proxy* também é o único canal por onde os processos recebem mensagens providas de fora do ambiente. Toda mensagem recebida pelo proxy é identificada pelo enreço lógico do processo destinatário. Cabe ao proxy converter este enedeço lógico em seu endereço físico e encaminhar a mensagem ao destino.

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

4.2.3 Tabela de endereços de processos

A tabela de endereços dos processos 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 de dos 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 cada migração, o *proxy* referente ao ambiente que recebeu o processo que migrou responsável por enviar mensagem aos demais ambientes, atualizando seu endereço do processo em questão.

Além de armazenar dados refrente aos endereços lógicos e físicos dos processos, a tabela de endereços armazena também uma variável de *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 no buffer de mensagens do *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 pode ser encontrado na Seção 4.4.1.

4.3 O componente process

Um processo é uma unidade discreta 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 comunicaçàò transparente para o *framework*.

Mais detalhes sobre a comunicação entre processos e a distinção entre comunicação entre processos cohabitantes e não-cohabitantes são detalhados na seção 4.5.

4.3.1 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* extende 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 4.2. A representação dos *slots* onde serão inseridos os códigos, na forma de métodos abstrados é ilustrada na representação UML da Figura 4.3.

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 méto 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 onde, em um laço repetitivo, São retirados os eventos da fila de eventos futuros para a sua execução. Vale ressaltar que como descreve (ALVES; WALBON; 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).

A solução apontada por (ALVES; WALBON; TAKAHASHI, 2009) é a de, após realizar a execução de um evento da fila de eventos futuros, enviar uma mensagem para si mesmo agendando a execução do próximo evento. Isso permitiria a intercalação entre mensagens para executar um novo evento com mensagens externas, trazendo novos eventos, mensagens de sincronização, etc.

Cabe aqui ressaltar que a ação de retirar um evento da fila de eventos futuros

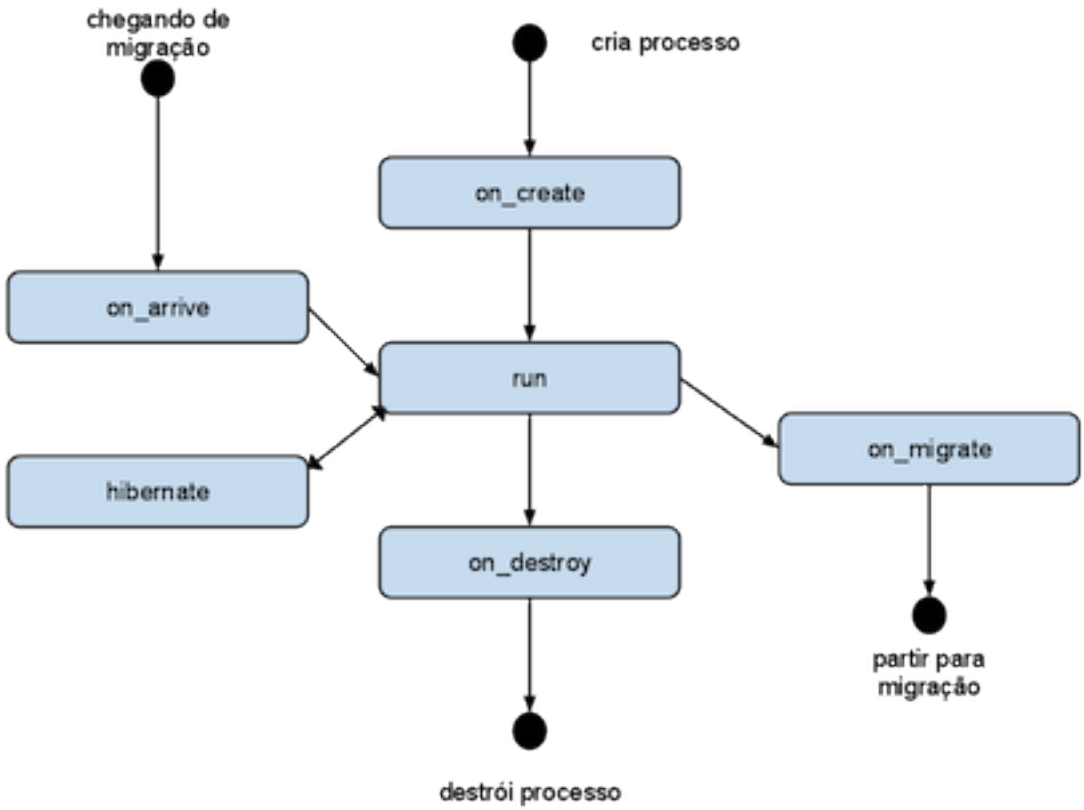

Figura 4.2: Ciclo de vida de um processo lógico.

é implementada internarmente na estrutura do *framework*, e o seu usuário 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, invocado ao se destruir um processo lógico.

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.

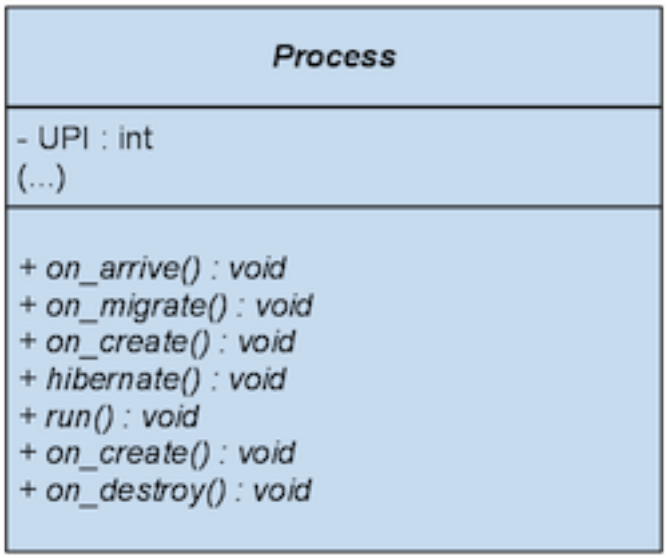

Figura 4.3: Representação em diagrama de classe do componente *process*.

4.3.2 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, a fim de se armazenar ou reutilizar posteriormente

Um processo deve possuir a propriedade de ser serializado quando conveniente. A serialização de um processo se dá internamente no *framework* a partir de chamadas do *middleware*. Ao usuário cabe garantir que todo o código escrito seja serializável.

4.4 Migração de processos

Uma das principais características propostas pelo *middleware* de comunicação que compões este trabalho á a possibilidade de que processos lógicos migrem de seu nó de origem para um diferente nó do sistema a fim de, por exemplo, prover um balanceamento de cargas através do sistema. Para que isso ocorra, naturalmente, 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 ocorre em uma determinada ordem, para garantir 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 estado do processo (a ser migrado) na tabela de endereços é modificade de Ativo para Trânsito.
2. O método on_migrate é invocado ainda no ambiente de origem do processo.
3. O processo a ser migrado é serializado pelo *environment* de origem e enviado para o ambiente de destino.
4. 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.
5. O ambiente de destino atualiza o estado do processo em sua tabela local de Ausente para Inativo.
6. 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.
7. O *environment* de origem atualiza o estado do processo para Ausente. Atualiza também o endereço físico do processo na tabela de endereços.
8. O processo executa o método on_arrive no ambiente de destino.
9. O *environment* muda o status do processo de Inativo para Ativo.
10. 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.
11. O processo, já em estado ativo, executa o método run.

12. Uma mensagem em *broadcast* é enviado a todos os *environment*, sinalizando o novo endereço físico correspondente à aquele processo acaba de migrar. As tabelas de endereços são então atualizadas.

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 se dá poque o processo foi serializado e todos os dados foram mantidos tais como estavam instantes antes da migração.

4.4.1 Atualização da tabela de endereços dos processos

Uma vez que um processe 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 difente 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 endereço atualizado do processo em questão), redireciona a mensagem ao proxy do ambiente que possui atualmente este processo. Porém, isso inclui mais um intermediário no processo de transmissão de mensagens. Sendo assim, duas ações, em momentos distintos, são efetuadas para garantir a atualização das tabelas de endereços de processos. Primeiro, o antigo *host* do processo, ao receber a mensagem, além de repassa-la ao atual *host* também devolve uma mensagem para o remetente da mensagem, notificando-o que o endereço do ambiente que contém o processo mudou, e atualiza este endereço.

Em um momento distinto, uma segunda ação de sincronismo de tabelas é disparada. Esta ação é iniciada de forma idependente 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 existente na simulação.

4.5 Troca de mensagens

No modelo de comunicação implementado pelo *middleware* de comunicação, a troca de mensagem 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 à 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.

4.5.1 Comunicação local direta

A comunicação direta ocorrem 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 P1 deseja enviar uma mensagem ao processo P2, o processo remetente possui o endereço lógico do processo destinatário. Este processo invoca um método interno de envio de mensagem que, por sua vez, invoca o *proxy* do sistema a fim de traduzir o endereço lógico em um endereço físico.

De maneira prática, ao se invocar o método de envio de mensagem do processo P1 este recebe internamente do *proxy* do sistema 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

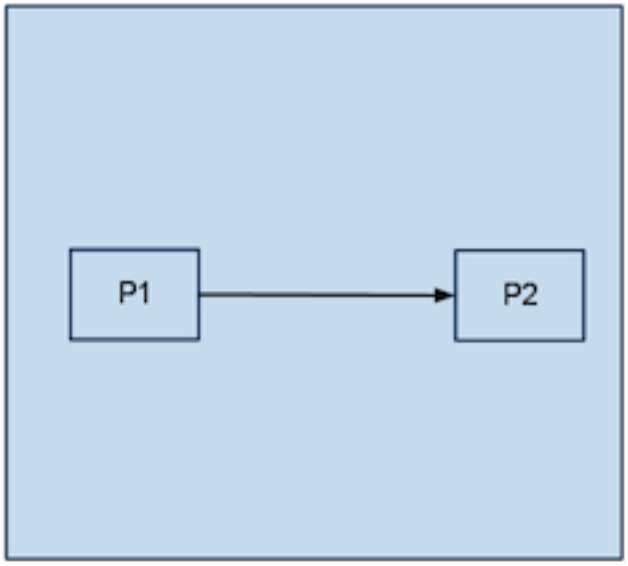

Figura 4.4: Comunicação direta *process-process*.

processo destinatário, e é capaz de enviar a mensagem diretamente para ele, sem uma nova intervençao do *proxy*.

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

4.5.2 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 4.5.

Quando o processo P1 deseja enviar uma mensagem ao processo P3, e estes habitam ambientes distintos no sistema, ao se executar o método para enviar uma mensagem, o *proxy* responsável pelo processo P1 identifica que o processo P3 não pertence ao mesmo ambiente. Assim, internamente o proxy devolve uma função ao processo P1 (de maneira idêntica ao que ocorre na comunicação direta), porém

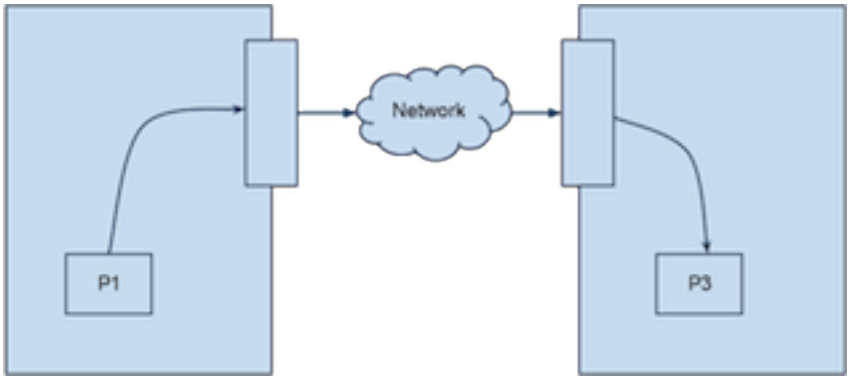

Figura 4.5: Comunicação indireta *proxy-process*.

com a diferença que esta função não possui o endereço físico do processo P3, mas sim o endereço físico do *proxy* responsável por P3 (este endereço é obtido através da tabela de endereços e do endereço lógico de P3).

Uma vez que o processo P1 recebeu (de maneira automática e interna ao *middleware*) o endereço físico do *proxy* responsável pelo processo P3, a mensagem é então enviada ao ambiente de destino e, uma vez entregue, é redirecionada pelo seu *proxy* para o processo P3.

A adoção da interferência dos *proxies* na transmissão das mensagens entre os processos pode parecer a princípio um ítem 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 migraçào de processos.

Em um modelo onde cada processo resolveria independentemente a tradução de endereço lógigo 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 proposta). Isso elevaria consideravelmente a quantidade de tabelas de endereço a se atualizar no caso de uma possível migração.

4.6 Comunicação grupal

Por definição agentes móveis não suportam comunicação grupal. Uma das lacunas de se utilizar agentes móveis para a implementação de um framework de simulação distribuída é justamente a ausência de um mecanismo nativo de comunicação grupal, necessário para a implementação do protocolo *Rollback* Solidário.

Uma alternativa para este problema seria o envio de múltiplas mensagens para cada *environment* presente no sistema, o que seria uma solução parcial, pois a medida que aumenta-se a quantidade de nós existentes no sistema, aumenta-se a quantidade de mensagens a serem enviadas.

A arquitetura do *middleware* proposto resolve este problema em dois níveis: primeiramente existe o envio de mensagens em broadcasta que atingem simultaneamente todos os *environments* do sistema. Em seguida cada environment envia uma mensage física local para cada processo. Como a comunicação entre *environment* e processo é local, não há sobrecarga na rede.

5 Arquitetura do framework de simulação

Uma vez definida a arquitetura do *middleware* de comunicação a ser utilizado pelo *framework* de simulação, este capítulo define a arquitetura e o funcionamento da camada de simulação, aqui genericamente entitulado *framework*.

Esta camada, conforme ilustrada na figura 5.1, compreende diversos sub-componentes do framework (aqui esses sub-componentes serão denominados genericamentes de módulos, para que se distingua dos objetos utilizados para descrever um modelo a ser simulado). Cada módulo que compreende o *framework* se liga ao módulo central, denominado *kernel*. Este 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).

Uma atenção maior é dado à biblioteca de componentes, que é a interface que o *framework* disponibiliza para o seu usuário descrever o modelo a ser simulado.

5.1 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 provido pela biblioteca de componentes devem ser utilizados em conjunto para descrever o comportamento do modelo, de maneira análoga a um componente eletrônico, que juntos

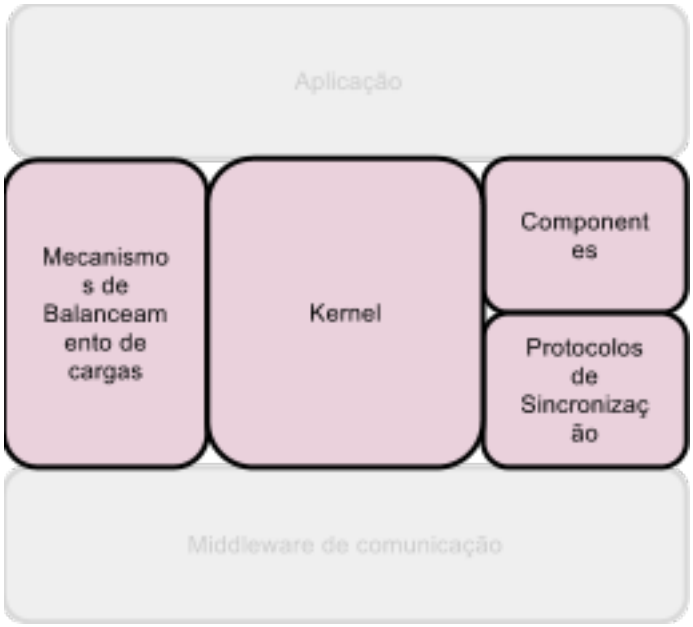

Figura 5.1: A camada aqui denominada *framework*.

compôem um circuito para realizar uma determinada ação. Cada componente possui suas características próprias 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 enxergarmos cada cliente que vai ao caixa do supermercado como um evento discreto, podemos 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 possuêm 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 componete gerador deve suportar parâmetros que, por exemplo, descreva a taxa de criação de novos evento, e as características de cada evento criado.

Os componentes são construidos diretamente em cima do objeto *process* da camada 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. Um exemplo disto é o caso em que um componente do tipo fila, que tem como características armazenar eventos que serão consumidos por um componente do tipo consumidor, seja encapsulado junto ao seu consumidor, evitando assim a possibilidade de que estes se separem, e diminuindo a quantidade de mensagens que trafegariam pela rede.

5.2 Componentes básicos

O tipo primitivo de um componente no *framework* proposto é desenvolvido sobre a classe *process* definida pelo *middleware* de comunicação (conforme ilustrado na figura 5.2) e deve ser capaz de proporcionar alguns elementos básicos para o controle do fluxo de eventos, como por exemplo um (ou diversos) canais de entrada de eventos, ambiente de processamento e canais de saída de eventos.

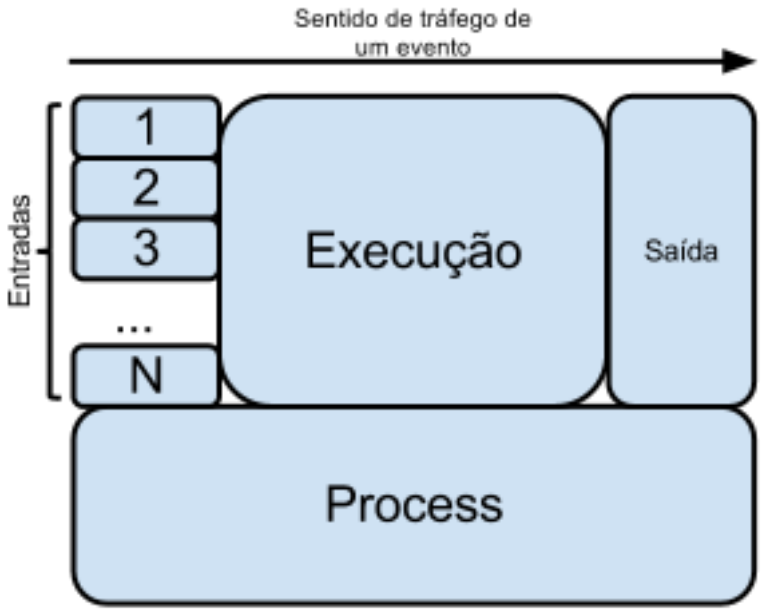

Figura 5.2: Arquitetura básica de um componente primitivo.

Sobre este componente primitivo são construídos quatro componentes básicos(Figura 5.3), suficientes para demonstrar a implementação de alguns modelos para simulação. Estes componentes são: fila, gerador, consumidor e divisor.

Por motivos naturais, nem todos os componentes implementos esse modelo em sua totalidade. Componentes 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.

5.2.1 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 retirando-se o primeiro elemento da fila, ou seja, o elemento com o menor *timestamp*.

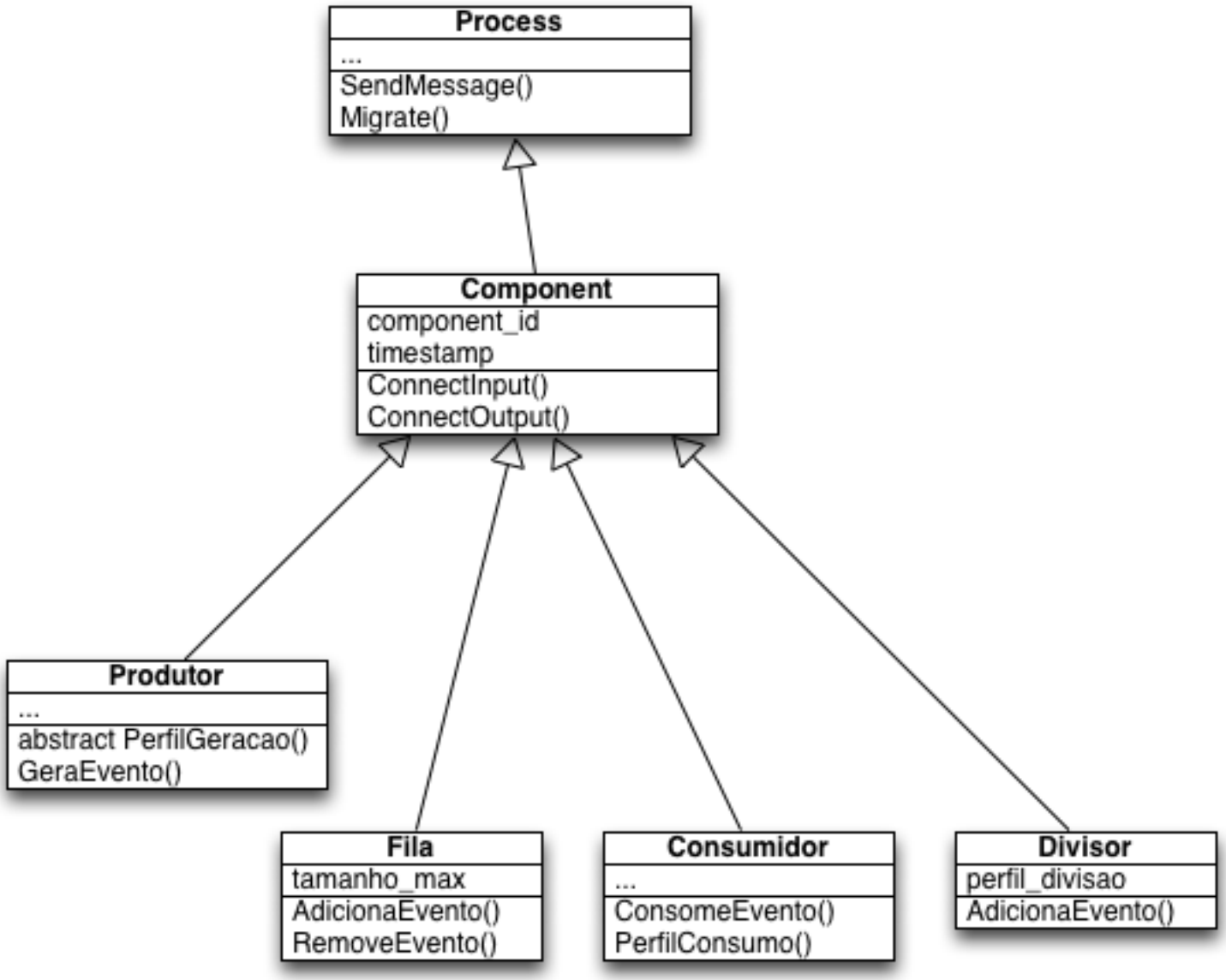

Figura 5.3: Hierarquia dos componentes básicos.

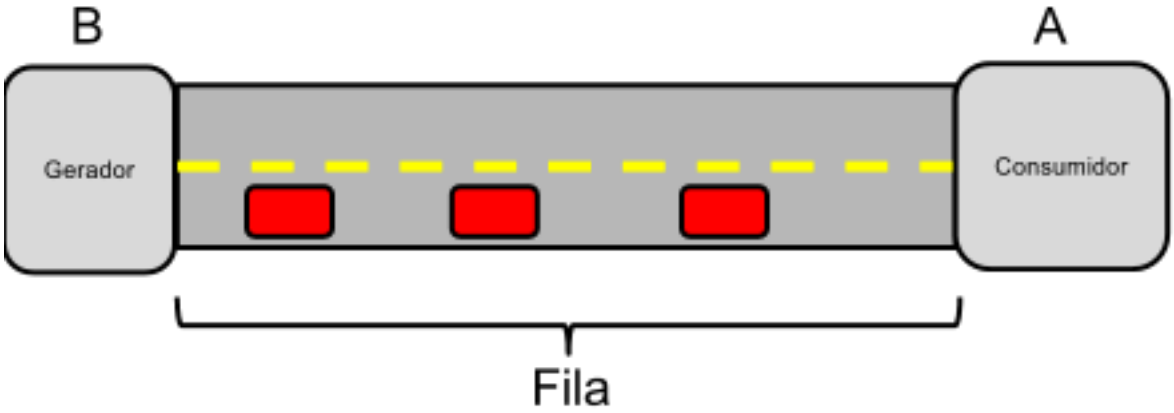

Figura 5.4: Modelagem de uma via urbana.

A fila é um componente passivo, ou seja, ela não executa nenhum tipo de ação sobre os eventos nela existente. 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* da fila.

Uma das características mais importantes 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 dois eventos, 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 5.4. 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 à 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.

5.2.2 O componente Gerador

Assim como ilustrado na Figura 5.4, 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 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.

Mais detalhes sobre o funcionamento das funções geradoras são abordados na seção **??**.

5.2.3 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. Outros dados levados em conta durante a execução de um evento por um consumidor são as características internas dos eventos. Esses parâmetros determinam, por exemplo, se o componente processará um evento de maneira mais

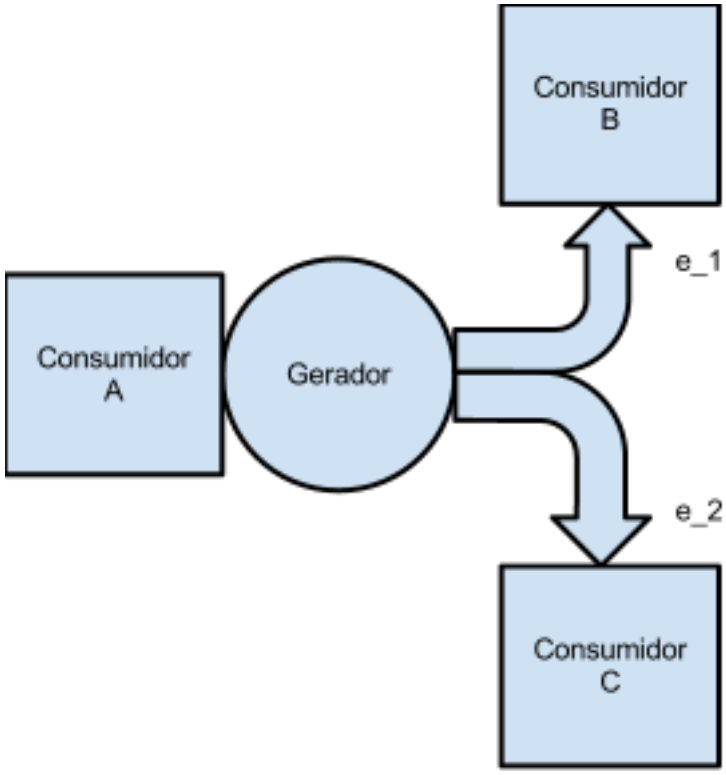

Figura 5.5: Um processo consumidor que gera eventos. No caso ilustrado, o consumidor A gera, através de um componente Gerador, simultaneamente os eventos e_1 e e_2 que são então encaminhados para os consumidores B e C

rápida ou mais lenta, qual a taxa de geração de novos eventos para si mesmo, entre outros.

O componente gerador possui múltiplas entradas de dados, porém apenas um canal de saída. Uma saída do gerador pode ser dividida em várias utilizando um componente divisor(ver subseção 5.2.4). A razão de se adotar este *design* de um único canal de saída e múltiplas entradas visa simplificar a arquitetura do modelo, uma vez que o componente gerador não precisaria se encarregar do comportamento que os eventos por ele processados tomariam após o processamento.

Uma vez que o evento é processado por um consumidor, este pode tomar três caminhos distintos: o evento pode simplesmente morrer, 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. Uma terceira possibilidade é que o evento sofra uma modificação nas suas características antes de ser encaminhado a diante.

Em condições especiais, um componente processador pode ser conectado à um gerador, e a ação de executar um evento poderia disparar a geração de um ou mais eventos distintos por esse gerador, que seriam então inseridos no sistema. Um exemplo prático desta aplicação é a simulação do fluxo de trabalho de um escritório onde a chegada de um documento pode disparar duas tarefas diferentes a serem feitas em paralelo (Figura 5.5).

5.2.4 O componente Divisor

Um componente divisor possui multiplas entradas e múltiplos canais de saída. A sua função é, dado um evento e que entra pelo divisor, aplicar uma função de decisão que mapeie este evnto para uma de suas saídas.

Assim como os componentes consumidor e gerador, o comportamento de um divisor é parametrizável através de funções que descrevem o seu comportamento com base nas características do modelo simulado.

A aplicação mais comum de um divisor é atrelado à saída de um processador, ou de um gerador, a fim de o evento que é entregue a ele seja redirecionado a um caminho aleatório, porém descrito por sua função de comportamento.

5.3 O kernel do framework

A parte central do *framework* desenvolvido neste trabalho aqui é ilustrada pelo módulo *kernel* (Figura 5.1). O *kernel* é o responsável por gerenciar a simulação, incrementando o *timestep* de cada componente, e tomando decisões de migração, comunicação, sincronismo e balanceamento de carga.

Para realizar tais funções o kernel se apoia diretamente no *middleware* de comunicação descrito no capítulo 4, além de utilizar os algorítmos de balanceamento de carga e os protocolos de sincronismo da simulação.

A cada incremento discreto da simulação, denominado *timestep*, o kernel do *framework* realiza as seguintes funções:

- Incrementar o *timestamp* de cada componente.
- Verificar se existem mensagens no *buffer* do middleware, e tentar redirecionalas 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 *straggler*. Caso afirmativo, designa ao módulo de sincronismo a tarefa de resincronizar o sistema.
- Gerencia, através do módulo de sincronismo, as tarefas de salvamento de estados.

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

5.4 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* possibilita uma facilidade quando se deseja trocar algum componente do sistema.

Para que isto seja possível, o *kernel* do sistema deve garantir uma *interface* de acessos a diveras 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, idendtificar mensagens *straggler*, além de ter acesso ao *middleware* de comunicação para trocar mensagens com os demais nós do sistemas, e requerer o *rollback* para um determinado *timestamp*.

5.5 Algoritmos de balanceamento de carga

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

Cabe ao *framework* prover uma interface que possibilite ao módulo de balanceamento requerere 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 algorítmo decide qual processo deve ser migrado, e para qual máquina.

Para que a migração seja possível, o *kernel* deve prover mecanismos para que o módulo de balanceamento de cargas requira a migração de um determinado processo.

6 Implementação

Para validar o conceito apresentado nos capítulos anteriores, a arquitetura do *framework* foi efetivamente implementada e testada.

Neste capítulo serão tratados detalhes de implementação do framework de simulação. Particularidades de implementação e decisões de *design* de software são expostos neste capítulo, assim como alguns exemplos de aplicação e utilização do framework.

Para esta implementação foi utilizada a linguagem *Python*. Toda a comunicação entre máquinas distintas foi feita utilizando *sockets* sobre o protocolo *TCP*. As tarefas assíncronas da implementação foram contruídas sobre a implementação de nativa *threads* provida pela própria linguagem.

6.1 A linguagem Python

Python é uma linguagem de programação poderosa e de fácil aprendizado. Possui estruturas de dados de alto nível eficientes, bem como adota uma abordagem simples e efetiva para a programação orientada a objetos. Sua sintaxe elegante e tipagem dinâmica, além de sua natureza interpretada, tornam Python ideal para scripting e para o desenvolvimento rápido de aplicações em diversas áreas e na maioria das plataformas.

O interpretador Python e sua extensa biblioteca padrão estão disponíveis na forma de código fonte ou binário para a maioria das plataformas (**??**), e podem ser

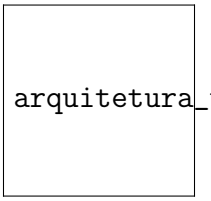

Figura 6.1: Diagrama de classes do *framework*.

distribuídos livremente. Também estão disponíveis distribuições e referências para diversos módulos, programas, ferramentas e documentação adicional, contribuídos por terceiros.

O interpretador Python é facilmente extensível incorporando novas funções e tipos de dados implementados em C ou C++ (ou qualquer outra linguagem acessível a partir de C). Python também se adequa como linguagem de extensão para customizar aplicações.

A linguagem vem sendo amplamnete empregada em desenvolvimento na área de computação científica devido ao seu desempenho satisfatório e sua facilidade de uso. Mais detalhes sobre a linguagem são tratados no apêndice A.

6.2 As camadas externas

A arquitetura apresentadas nos capítulos anteriores deste documento é representada pelo diagrama *UML* da figura 6.1 (Uma versão expandida, portanto mais completa, deste diagrama pode ser encontrada no anexo B). Vale ressaltar que esta representação é completamente independente da linguagem escolhida para sua implementação, porém atrelada ao paradigma de programação orientado à objetos, eleito para construir o *framework* de simulação.

A arquitetura da árvore de diretórios é ilustrada na listagem 6.1. A partir dela pode-se ilustrar a modularização e a divisão das módulos do framework. Os módulos são descritos a seguir:

Listing 6.1: "Árvore de diretórios do projeto."

```
|-- doc
|-- examples
|   |-- basic_simulation
|   |
|   |-- check_installation.py
|   |-- __init__.py
|   +-- simtool
|-- README
|-- setup.py
+-- t100
    |-- components
    |   |-- components.py
    |   +-- __init__.py
    |-- core
    |   |-- algorithms
    |   |   |-- __init__.py
    |   |   +-- step_algorithms.py
    |   |-- base_components.py
    |   |-- blah.py
    |   |-- errors.py
    |   |-- __init__.py
    |   +-- simulator.py
    |-- __init__.py
    |-- simtool
    |   |-- distributed
    |   |   |-- dummy_environment.py
    |   |   |-- example_of_use_proxy.py
    |   |   |-- __init__.py
```

```
|   |   +-- proxy.py
|   |-- distributed_env.py
|   |-- __init__.py
|   |-- parallel
|   |   +-- __init__.py
|   |-- parallel_env.py
|   |-- sequential_env.py
+-- test
    |-- __init__.py
    +-- tests.py
```

- **doc:** Contém toda a documentação do projeto, inclusive esta monografia.
- **examples:** Contém exemplos de aplicação do *framework*.
- **setup.py:** *Script* de instalação do *framework*.
- **t100:** Código fonte do *framework*.

Nas seções seguintes são apresentados detalhes internos da estrutura do código implementado.

6.2.1 Módulos internos do *framework*

O código fonte do *framework* pode ser dividido inicialmente em quatro grandes módulos:

- **components:** Possui toda a descrição dos componentes utilizáveis na simulação.
- **core:** O módulo *core* possui todos os principais algorítmos internos ao *framework*. São os algorítmos não-modularizáveis do projeto.

- **simtool:** O módulo *simtool* possui as implementações de algorítmos modularizáveis do projeto. Neste caso entram algorítmos de sincronismo, de troca de mensagens, etc.

- **test:** São os *scripts* de testes do sistemas, necessários para validar a integridade do projeto após eventuais mudanças no código.

O módulo *core* possui as especificações mínimas que descrevem componentes e o *kernel* do simulador. São encapsulados também neste nível dados algumas classes de manipulação de erros e algoritmos básicos de execução.

O módulo *components* possui a implementação base necessária para se descrever um componente para a simulação, além de componentes genéricos. Mais detalhes sobre este módulo são tratados na seção 6.5.

6.3 Implementando a comunicação

A linguagem de programação *Python* oferece uma ampla e funcional biblioteca padrão, que provê, de forma nativa na linguagem, métodos para implementação de *threads, sockets*, persistência de dados e estruturas de dados.

Conforme ilustrado na figora 6.2, o proxy mínimo de comunicação deve oferecer os seguintes métodos: *send, receive, listen.*

Para a implementação do *proxy* de comunicação

6.4 O objeto Message

Toda a comunicação entre instâncias de *proxy* é realizada enviando e recebendo objetos do tipo *Message*. Um objeto do tipo *Message* é um objeto obrigatoriamente serializável (ou seja, ele pode ser convertido de objeto para um *stream* de bytes a

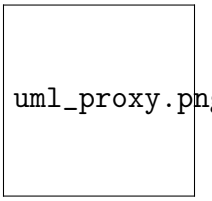

Figura 6.2: Diagrama de classes da classe *proxy*.

qualquer instante do seu ciclo de vida. Este *stream* de *bytes* deve ser usado para reconstruir o objeto *Message*).

O objeto possuim os campos *content*, que é o conteúdo da mensagem propriamente dito, além dos campos de endereço lógico do remetente e do destinatário. Cabe ao *proxy* converter este endereço lógico em seu correspondente físico antes da transmissão.

6.4.1 O método *send*

O método *send* recebe como parâmetro um objeto do tipo *Message* e o envia para o destinatário designado a ele. O próprio *proxy* deve ser capaz de, em cooperação com o *kernel* do *framework*, resolver o endereço lógico do destinatário para seu correspondendte endereço físico, e realizar a comunicação.

6.5 Implementando os Componentes

6.6 Implementando o Environment

6.7 Exemplos de aplicação

```
import sys

def primes(n):
```

```
    for i in xrange(2,n):
        for j in xrange(2,i):
            if i%j == 0:
                break
        else:
            yield i

if __name__=='__main__':
    v = int(sys.argv[1])
    prm = [p for p in primes(v)]
```

7 Discussões finais e Conclusões

APÊNDICE A – A linguagem Python

APÊNDICE B – Diagrama de classes

APÊNDICE C – Distribuiçao

O projeto *framework* desenvolvido neste trabalho recebeu o nome de T100. O projeto T100, incluindo seu código fonte e toda a documentação é distribuído sobre licensa *GPL - General Public License*, que dá ao seu usuário garantias de liberdade para uso, modificação e redistribuição do software como um todo, incluindo o seu código fonte.

C.1 Licensa

O projeto T100 é distribuído sobre licensa *GPL v3.0*.

C.1.1 *License*

Listing C.1: "Licensa de uso do software"

```
Antonio Ribeiro Alves
Copyright (C) 2012  Antonio Ribeiro Alves

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.
```

C.1.2 Informações sobre a licensa

Para mais informações sobre a lincesa *GPL v3.0* visite:

http://www.gnu.org/licenses/gpl.html

C.2 Disponibilidade

O código fonte, junto aos exemplos de uso e documentação deste projeto, podem ser adquiridos através do endereço:

http://github.com/alvesjnr/T100

Referências Bibliográficas

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

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>.

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

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.

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

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.

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.

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.

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

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

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.