Antonio Ribeiro Alves Júnior

Arquitetura de um Framework para Simulação Distribuída

Itajubá - MG

01 de Junho de 2012

Antonio Ribeiro Alves Júnior

Arquitetura de um Framework para 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 Junho de 2012

Sumário

Lista de Figuras

1 Introdução

1.1 Simulação de eventos discretos

1.2 Sistemas centralizados e distribuídos

1.3 Simulação distribuída de eventos discretos

2 Simulação Distribuída de Eventos Discretos

2.1 Categorias de protocolos de simulação

2.2 O protocolo Time Warp

2.3 O protocolo Rollback Solidário

2.4 Balanceamento de cargas

3 Proposta deste projeto

Neste capítulo pretende-se apresentar a proposta de uma arquitetura que visa sustentar o desenvolvimento de aplicações de simulação distribuída de maneira transparente para o usuário final. Em complemento é trazido também neste capítulo algumas soluções já existentes e, por fim, pretende-se defender a proposta de se optar por uma nova arquitetura de comunicação entre processos lógicos.

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, dentre outros, acarretam na existência de diversas tarefas paralelas à simulação em si, que devem ser gerenciadas pelo desenvolvedor que pretende implementar a simulação.

Soma-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)). Obtemos neste ponto um cenário onde o desenvolvedor acaba tendo de se ater a diversos detalhes alheios à simulação propriamente dita, que torna impraticável um desenvolvimento sustentável de simulações distribuídas.

Assim como proposto em (CRUZ, 2009), um framework de simulação tem o ob-

jetivo 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 diversas tarefas vitais para a simulação, deixando-o focado apenas em sua modelagem e execução. Para prover tal separação entre a aplicação escrita pelo usuário e o *framework*, este compromete-se a prover três funcionalidades essesciais para garantir tal isolamento: encapsulamento, transparência e reusabilidade.

3.1.1 Encapsulamento

Uma das funções de um *framework* é o de encapsular diversos elementos que não tratam diretamente da simulação, porém sustentam funcionalidades que dão vida à simulação propriamente dita. Exemplo disso é a possibilidade de se criar elementos lógicos para compor o modelo, análogos à componentes em um circuito elétrico. Quando encapsulamos o funcionamento de uma fila ou o de um processo consumidor em uma classe, por exemplo, estamos isolando sua implementação e seus detalhes do usuário final do *framework*. A este usuário cabe reutilizar estes componentes e, quando julgar necessário, criar componentes baseados nesses primitivos.

Outro ocasião em que pode-se 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 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 baseia 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: sincronismo, comunicação, migração e balanceamento de carga em um sistema distribuído de simulação), deixando ao usuário apenas a função de desenvolver a aplicação sem a necessidade de se preocupar com questões que estão fora de seu domínio.

O conceito de transparência neste caso remete-se que a intenção do *framework* 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 outros.

3.1.3 Reusabilidade

Ao se eleborar um *framework*, o responsável pelo seu *design* deve permitir que componentes de interesse 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 deste *framework*.

No caso da simulação distribuída, componentes como protocolo de sincronização ou o sistema de balanceamento de cargas poderiam ser plugáveis, o que permitiria a reutilização do mesmo código de simulação no mesmo *framework*, porém

com características diferentes. Isso leva à uma reutilização de código, economizando no desenvolvimento 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 a carga 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 ao mesmo tempo que provém eficientes mecanismos para gerenciar a troca de mensagens entre os processos, 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) baseia-se nos agentes móveis para se desenvolver a aplicação de simulação distribuída.

Tanto nos casos que utilizam agente móveis quanto nos casos que baseiam-se em suprir soluções para troca de mensagens não é encontrada nativamente todos os mecanismos necessários para a implementação de um sistema de simulação distribuída que suporte tanto troca de mensagem quanto migração de processos. Ao utilizar-se de agentes móveis, que possuem tanto mecanismo de mobilidade

quanto mecanismos de troca de mensagens, obtemos em um primeiro momento todos esses elementos. Porém, ao se mover um objeto de um nó para outro, perde-se a referência que havia deste objeto, forçando a se implementar um mecanismo que corrija este cenário. Vale citar também que mecanismos como comunicação grupal, essencial na implementação do *Rollback* Solidário não é compreendido por agentes móveis.

Visando isso este trabalho se propões em apresentar uma solução para se desenvolver simulações distribuídas de eventos discretos que encapsule os macanismos de sincronização de processos, balanceamento de carga, *design* de componentes para a construção do modelo a ser simulado.

Para que seja sustentada tanto os mecanismos de sincronização quanto o balanceamento de carga, é vital que o *framework* supra também as necessidades básicas de comunicação, troca de mensagens, serialização de processos lógicos, migração destes processos e comunicação grupal. Estas funcionalidades são providas pelo *middleware* de comunicação do *framework*. Uma característica fundamental do *middlere* 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 fim de suprir todas as necessidade de um sistema que suporte plenamente a migração de processos lógicos, é apresentado na figura 1 uma visão bastante ampla da arquitetura do framework de simulação distribuída aqui proposto.

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.

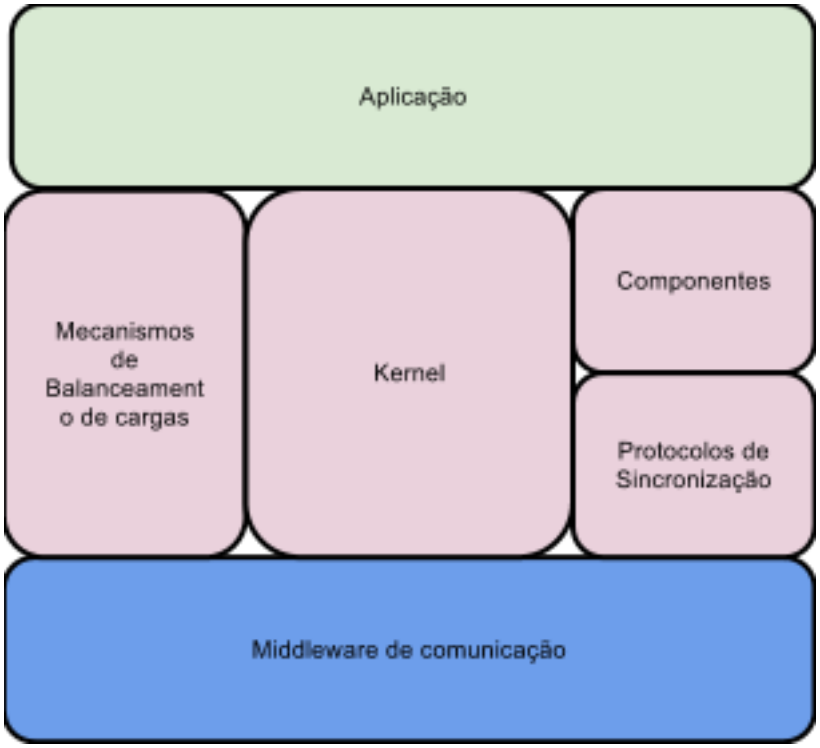

Figura 1: Camadas da arquitetura do *framework*.

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, processos consumidor, gerador de eventos, etc.

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, reenvio de mensagens para destinatários que migraram, gerenciamento de recursos do sistema e mecanismo de comunicação grupal.

Conforme descrito na seção 3.2, os mecanismos de troca de mensagens existentes não contemplam nativamente os requisitos que possibilitem tanto a troca

de mensagem entre processos e sua migração e, ao mesmo tempo, opere de forma transparente quanto à referenciação de processos lógicos que migraram. Nos casos que mais se aproximam dos requesitos do sistema, os agentes móveis, há uma transparência na troca de mensagens até o momento em que um agente migra para um nó distinto do sistema. Ao se mover para um ambiente diferente do seu ambiente de origem, implementações de agentes móveis como *Aglets* carecem de uma intervenção do usuário para que se refatore os valores dos endereços físicos dos objetos. O middleware aqui proposto visa resolver este problema, tornando a comunicação entre processos completamente transparente para o seu usuário.

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 explicitar 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

Este capítulo descreve em detalhes a arquitetura do proposto *middleware* de comunicação, tais como suas partes e suas principais funções, como troca de mensagens, migração e serialização.

4.1 Componentes do middleware

A arquitetura proposta é dividida em dois componentes principais: o ambiente (*envinonment*) e o processo lógico (*process*). Um ambiente representa um nó lógico 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 componente da estrutura do *middleware*, 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. Por sua vez, um processo somente pode estar contido por um único *environment* em um determinado momento.

Assim, sendo p um processo lógico e e um ambiente de simulação contento n processos, então definimos:

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

$$e_k = \{p_i | i < n\} \tag{4.1}$$

Definição 2: Um processo só pode estar contido em um único ambiente em um instante bem definido

$$p_k \in e_l \rightarrow \nexists e_m, p_k \in e_m, e_l \neq e_m \tag{4.2}$$

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 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 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*.

4.2 O componente environment

Essencialmente, um ambiente 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 a existência de uma camada que seja responsável por gerenciar o ciclo de vida de processos, como criação, migração e destruição de processos.

4.2.1 Estrutura interna

Internamente, o *environment* apresenta as seguintes estruturas básicas (conforme ilustrada na Figura 2):

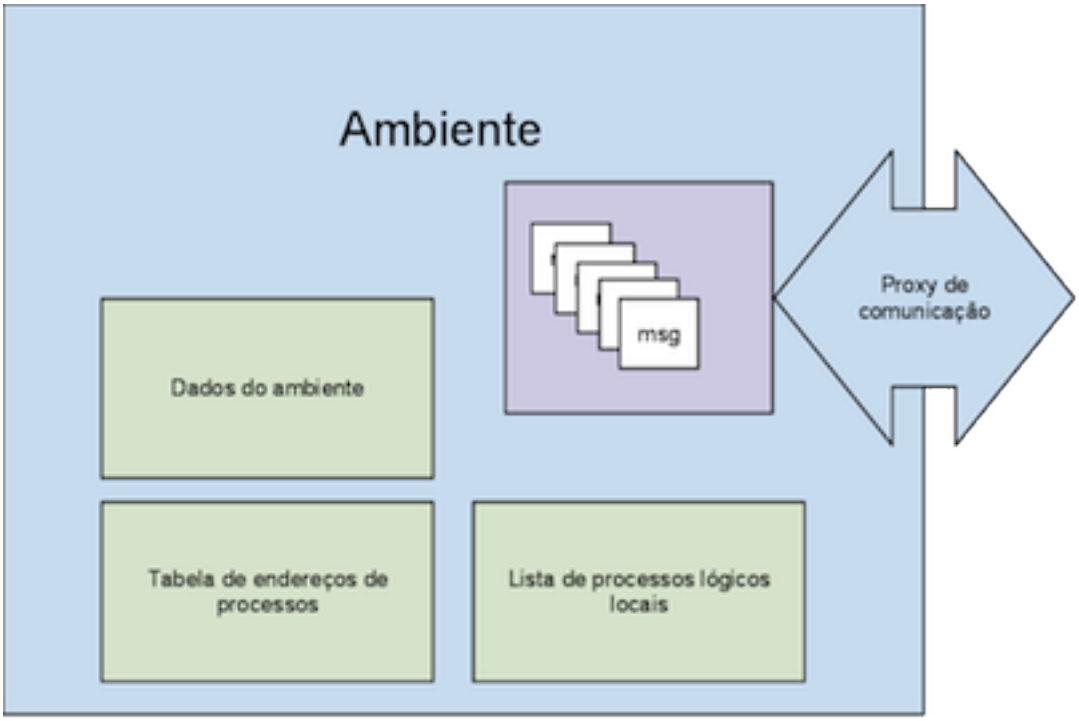

Figura 2: A arquitetura interna de um *environment*.

- 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) do ambiente na rede, nome 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 que o representa no sistema de simulação.

A comunicação entre dois *environmets* se dá através seu endereço físico na rede. Tal inflexibilidade é justificada quando assumimos que um *environment* tem todo o seu ciclo de vida atrelado a uma mesma máquina física.

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 noem lógico do processo destinatário. Cabe ao proxy converter este nome lógico em seu endereço físico e encaminhar a mensagem ao processo de destino.

Como todo o tratamento de envio e recebimento de mensagens deve ser não-blocante, este componente deveoperar de maneeira assíncrona aos demais componentes do *environment*. De maneira análoga, pode se dizer que o *proxy* é um subprocesso executando deentro do processo físico *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 sim dados de todos os processos existentes na simulação (mesmo que desatualizados, dependendo do instante). Dados referentes aos processos residentes no mesmo *environment* da tabela de endereço de processos estarão, invariavelmente, atualizados.

Conforme previamente mencionado, para realizar a comunicação entre processos é utilizado como identificador do processo não o seu endereço físico, mas sim seu endereço lógico no sistema. O motivo de se utilizar o endereço lógico é que este, ao contrário do endereço físico, é único em todo o ciclo de vida do processo. Isto garante que após uma migração de um environment para outro, um processo possa continuar a ser encontrado pelo mesmo endereço lógico (o que não aconteceria caso o endereço físico fosse usado, uma vez que após uma migração o endereço de IP do environment, e a porta a qual o processo está vinculado, sofreriam variações). Esta característica de ser identificado sempre pelo mesmo endereço lógico é fundamental para evitar a constante alteração de dados referentes aos processos em inúmeras partes do sistema.

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 foi migrado para um ambiente diferente, porém ainda não atualizou a tabela de endereços com seu endereço 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*. Este deve ser capaz de abrigar componentes que descrevem o comportamento do sistema a ser simulado, apoiando-se nos mecanismos de troca de mensagens, serialização e locomoção (migração) de processos providos pelo *environment*.

Um processo lógico possui três identificações distintas no sistema: seu endereço físico em memória, seu endereço físico na rede e seu endereço lógico no sistema de simulação. Na arquitetura deste *middleware*, o processo é referenciado para troca de mensagens ou por seu endereço físico na rede, ou por seu endereço lógico no sistema de simulação. A distinção se a comunicação é feita através do endereço físico ou lógico se dá a partir de qual nível da arquitetura se dá a comunicação ().

Como uma das premissas do *middleware* de comunicação é tratar de maneira completamente transparente para o usuário a comunicação entre os processos, o mecanismo de troca de mensagem entre dois objetos do tipo *Process* deve ser encapsulado e resolvido pelo próprio objeto (em conjunto com o *environment* e seu *proxy*), não transferindo assim ao usuário os encargos da troca de mensagem entre processos.

Neste ponto a comunicação se divide em doi 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 comunicaçào transparente para o *framework*.

Desta forma, internamente um processo lógico deve ser capaz de se comunicar com o processo alvo da maneira mais eficiente possível. Assim sendo, é natural que processos que residam em um mesmo *environment* possuam um canal diferenciado de comunicação, ao ponto que processo que estão em ambientes distintos compartilhem canais comuns de comunicação.

Em contrapartida, é justificável que processos que não se comunicam com grande frequência (ou que nunca se comunicam) não tenham conhecimentos uns dos outros. A abstração provida pelo *environment* garante o isolamento entre o endereço físico de um processo e o seu nome lógico no sistema. Este desacoplamento entre endereço lógico e físico de um processo garante também a continuidade da comunicação entre processos lógicos após sua migração. Assim sendo, uma vez que um determinado processo tenha que enviar uma mensagem para um segundo processo, cabe ao *environment* identificar o destinatário da mensagem e a despachar.

Toda comunicação que é iniciada a partir de um processo, é feita através do endereço lógico do processo. Isto garante que o usuário do *middleware* não precise se adentrar em estruturas físicas do sistema, como saber qual o endreço físico (IP:PORT) está o processo com o qual ele deseja se comunicar. Cabe ao *environment* que abriga o remetente da mensagem encaminhá-la para o endereço físico correspondente.

O endereço lógico de um processo é denominado *Unique Process Identifier - UPI*. Para garantir que cada processo tenha um endereço lógico único no sistema sem a necessidade de sincronização de todo o sistema em cada criação de um novo processo lógico, a construção deste endereço lógico se dá por uma tupla,

onde o primeiro elemento é o *UEI* do *environment* onde este processo foi criado, e o segundo elemento da tupla é um número sequencialmente incrementado pelo ambiente a cada novo processo criado. Isto garante que cada processo possua um par de números único em todo o sistema compondo a sua tupla de identificação.

Vale ressaltar que o *UPI* é único em todo o ciclo de vida de um *process* e, portanto, não se altera em caso de migração. Mesmo após migrar de um ambiente para outro, o primeiro elemento da tupla de identificação continua representando o ambiente onde o processo foi criado pela primeira vez.

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, *slots* onde serão implementados trechos importantes para fases específicas da vida do processo. Essas fases estão ilustradas no diagrama da Figura 3. A representação dos *slots* onde serão inseridos os códigos, na forma de métodos abstrados é ilustrada na representação UML da Figura 4.

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 a objetos. Imediatamente após executar o código contido no método on_create, o próximo méto invocado automaticamente pelo sistema é o método run.

O conteúdo do método run é onde o corpo da simulação deve estar implementado. É neste ponto do código onde a a simulação deverá retirar o próximo evento

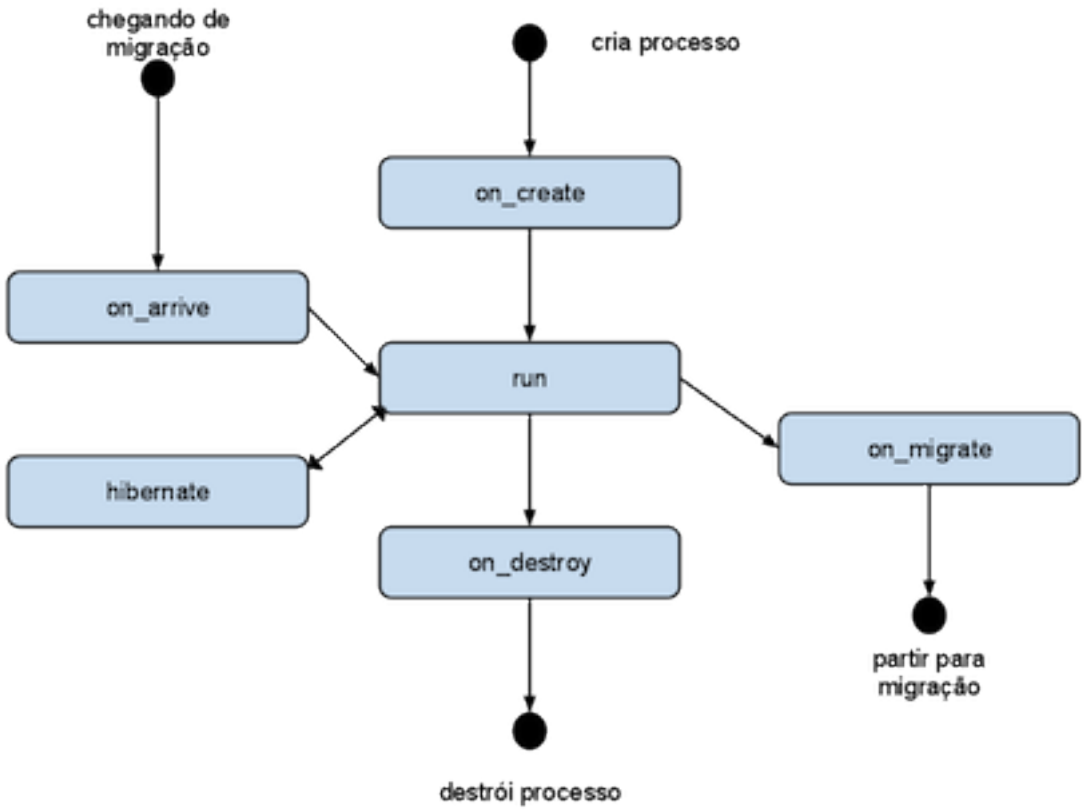

Figura 3: Ciclo de vida de um processo lógico.

da lista de eventos futuros e executá-lo. Vale ressaltar que como descreve (ALVES; WALBON; TAKAHASHI, 2009), o processo de se retirar eventos da fila de eventos futuros para execução não pode se dar na forma de um laço de repetição, pois tornaria a execução do método run não-preemptiva. A solução apontada por (ALVES; WALBON; TAKAHASHI, 2009) é a de, após realizar a execução de um ítem da fila de eventos futuros, enviar uma mensagem para si mesmo agendando a execução do evento seguinte. Isto garante a preempção da simulação para tratamento de mensagens externas, migrações, etc.

Os demais métodos invocados automaticamente pelo sistema são on_migrate,

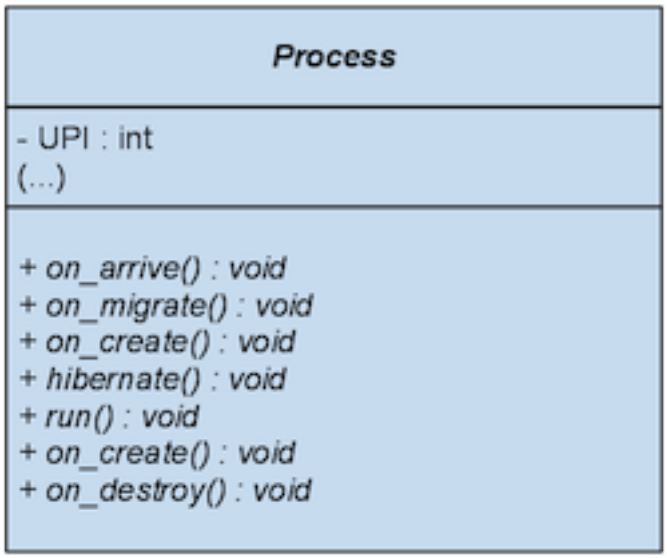

Figura 4: Representação em diagrama de classe do componente *process*.

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 o 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.

4.3.2 Serialização de um processo

Serialização é a capacidade que um objeto possui de converter sua estrutura interna de dados em um formato estático, a fim de se armazenar ou reutilizar posteriormente

Um processo deve possuir a propriedade de ser serializado quando conveniente. Um processo não possui a capacidade de se serializar, ou de serializar um outro processo. O processo de serialização de um processo se dá somente a partir de chamadas internas 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 nó distinto no sistema. Para que isso ocorra, naturalmente, o nó destinoo 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 deve ocorrer a fim de sinalizar

1. O estado do processo 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.

Vale ressaltar que uma vez que esteja em processamento (executando o método run), o processo só executa a migração após o término da execução do evento em questão. Sendo assim, o método run deve ser implementado de maneira a possibilitar a preempção de eventos.

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 possuem os dados atualizados. Todo ambiente difente dos envolvidos no processo de migração possuem em suas tabelas de endereços de processos dados incorretos quanto à sua localização, portanto, enviariam mensagens para o ambiente antigo, ao qual o processo destinatário não mais pertence.

Ao receber uma mensagem de um antigo hospedeiro, um *environment* (que possui o seu novo endereço), redireciona a mensagem ao proxy do ambiente que possui atualmente o processo em questão. 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 baseado em *proxy*, quando um processo deseja enviar uma mensagem a um segundo processo, a mensagem é primeiramente enviada ao *proxy* do *environment* ao qual o processo remetente reside, e o proxy é responsável por resolver a correlação entre endereço lógico e o endereço físico, e enfim enviar a mensagem ao processo destinatário. O principal motivo de se deixar o proxy responsável pelo envio da mensage é se possibilitar que existam uma pequena quantidade de tabelas de correlação de endereços, e assim, que existam menos atualizações de tabelas em uma migração.

Em um modelo onde cada processo resolveria o envio de mensagem diretamente, a quantidade de tabelas de endereços seria proporcional à quantidade de processos (e não proporcional à quantidade de ambientes, como é na arquitetura proposta). Ao haver uma migração, existirão menos tabelas a serem atualizadas na arquitetura baseada em comunicação inter-proxies, ao passo que na comunicação direta entre processos, a quantidade de tabelas a serem atualizadas seria bem maior.

4.5.1 Comunicação direta e 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 passa através do *proxy* do *environment* do remetente, e esse é responsável por enviar a mensagem ao *proxy* do processo destinatário. O fluxo de comunicação é ilustrada de maneira simplificada na Figura 5.

Neste caso, quando ao processo P1 enviar uma mensagem ao processo P3, o próprio processo requisita diretamente ao *environment* o envio da mensagem. A mensagem é então encaminha de P1 para o *proxy* do seu *environment* que detecta, através da tabela de endereço de processos, o endereço físico do processo alvo.

Uma vez que a mensagem a ser enviada está em posse do *proxy* do *environment* destinatário, duas modalidades de comunicação podem ocorrer. A modalidade padrão é a comunicação indireta, onde o *proxy* em posse da mensagem irá envia-la ao *proxy* do ambiente que contém o receptor da mensagem. Uma vez que a mensagem esteja no *proxy* do destinatário, este a encaminha para o processo (Figura 5).

Uma alternativa à comunicaçào indireta é a comunicação direta, onde o *proxy* do processo de origem, uma vez obtido o endereço físico do processo destinatário, envia a mensagem diretamente para o processo receptor (Figura 6). O método direto é útil por eliminar um intermediário na transmissão da mensagem, mas possui um inconveniente que pode anular o ganho da eliminação do intermediário. Caso haja uma migração, a mensagem não é automaticamente redirecionada para o novo ambiente em que o processo se localiza, mas sim é levantado uma excessão na comunicação, o que levaria a um tratamento de excessão, que por sua vez seria encarregado de localizar o processo em seu novo endereço físico e retransmitir a mensagem.

Cabe salientar que em caso de falha na transmissão da mensagem na modalidade de comunicação direta, a excessão levantada deve ser tratada não pelo

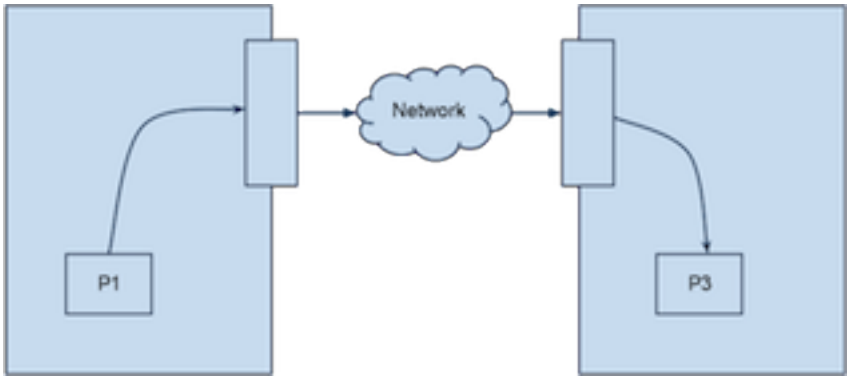

Figura 5: Comunicação indireta *proxy-process*.

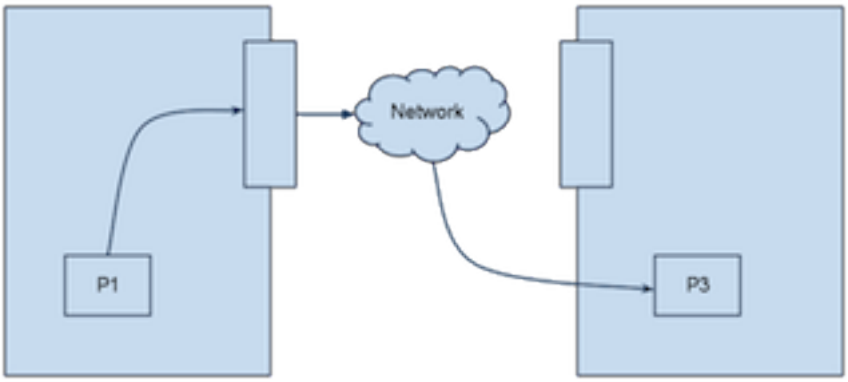

Figura 6: Comunicação direta *proxy-process*.

processo, mas pelo *environment*, pois a mensagem encontra-se em posse do *proxy*

4.5.2 Comunicação local direta

Uma terceira modalidade de comunicação é a comunicação direta, via endereço físico (Figura 7), entre processos que habitam um mesmo ambiente. Nesta modalidade um processo que se comunica constantemente com outro processo no mesmo *environment* adquire o seu endereço físico e faz a comunicação direta, sem a necessidade de se passar por um *proxy* de comunicação.

Assim como na comunicação direta, neste caso há um ganho na transmissão da mensagem entre origem e destino por se excluir o *proxy* como intermediário

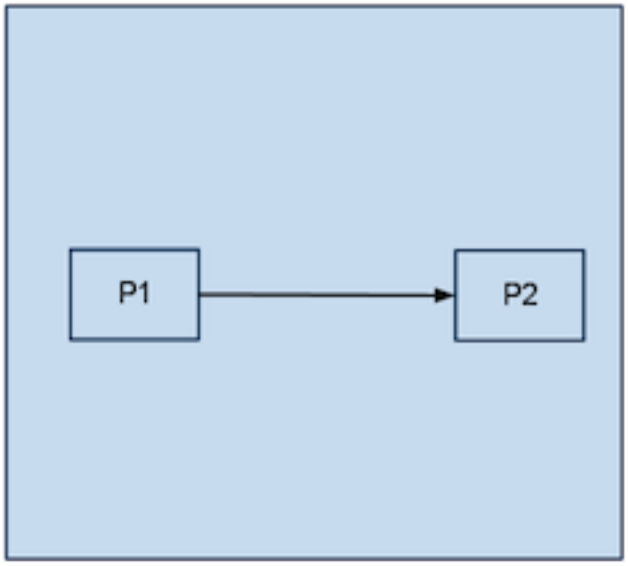

Figura 7: Comunicação direta *process-process*.

da transmissão. Entretanto, no caso de uma migração, as mensagens não seriam automaticamente redirecionadas para o processo destinatário, cabendo assim ao usuário do *middleware* o tratamento de excessão caso esta aconteça.

A comunicação local direta, assim como a comunicação direta, são opções que devem ser exploradas em casos onde a migração de processos é reduzida ou nula.

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 durante este projeto, o próximo passo é a definição do funcionamento e da arquitetura da camada de simulação, aqui genericamente denominado *framework*.

Esta camada, conforme ilustrada na figura 8, compreende diversos sub-componentes do framework (aqui esses sub-componentes serão denominados genericamentes de módulos, para que se distingua dos componentes, 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).

5.1 O módulo Componente

Um componente é uma abstração de um comportamento que desejamos reproduzir na nossa simulação. Na abstração de um sistema de caixa de supermercados, por exemplos, pode-se extrair três componentes (que são comuns em muitas situações na simulação de eventos discretos): o produtor, a fila de espera e o consumidor. O produtor seria o encarregado por criar clientes e envia-los à fila de espera para que o conumidor, que representa o caixa do supermercado, execute cada evento.

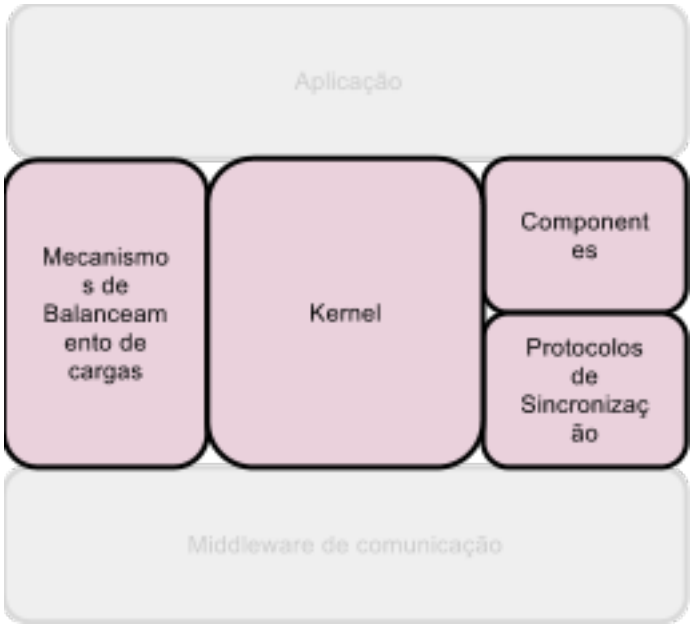

Figura 8: A camada aqui denominada *framework*.

Uma biblioteca de componentes deve proporcionar objetos parametrizáveis que permitam a descrição do comportamento de cada um deles. No caso citado anteriormente, o componete produtor 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 são processo cohabitantes ou não-cohabitantes.

Em alguns casos é conveniente que componentes sejam combinados a fim de que um novo componente seja criado. Uma das justificativas seria, por exemplo, garantir que dois componentes primitivos se comportem como um único compoente, evitando assim que estes sejam por ventura separados e passem a cohabitar

ambientes diferentes.

Em um exemplo, é conveniente que o 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 em cima da classe *process* (conforme ilustrado na figura 9) 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.

Sobre este componente primitivo são construídos quatro componentes básicos(Figura 10), 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 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

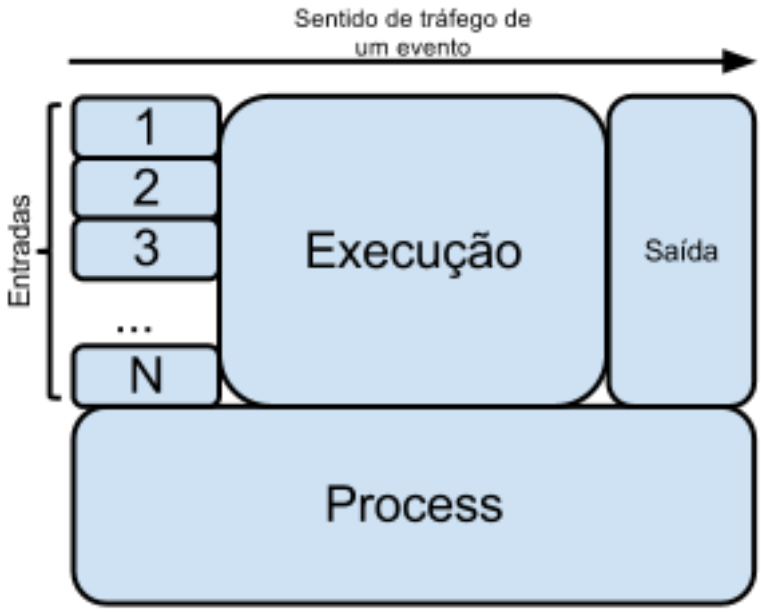

Figura 9: Arquitetura básica de um componente primitivo.

timestamp.

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 àquela 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.

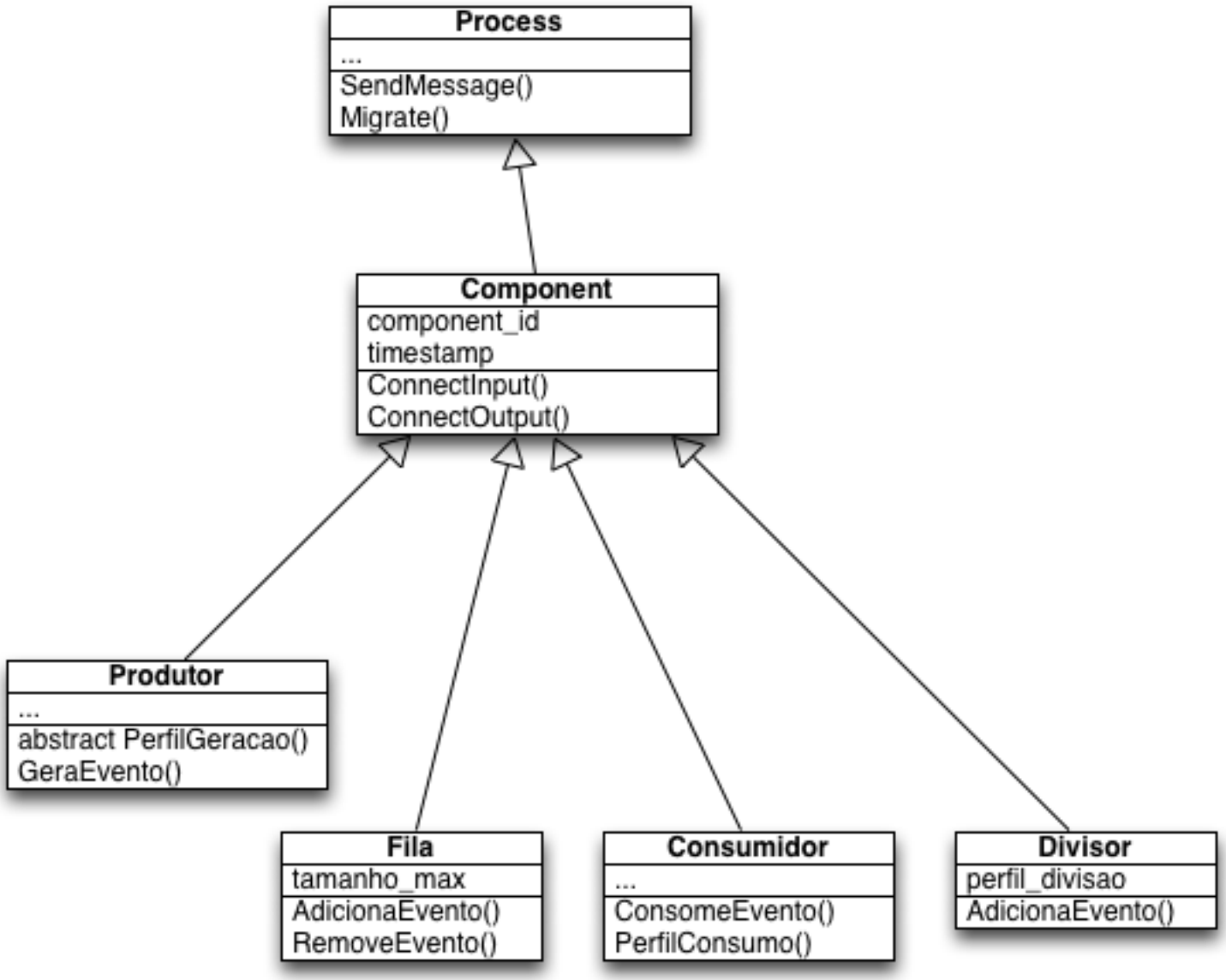

Figura 10: Hierarquia dos componentes básicos.

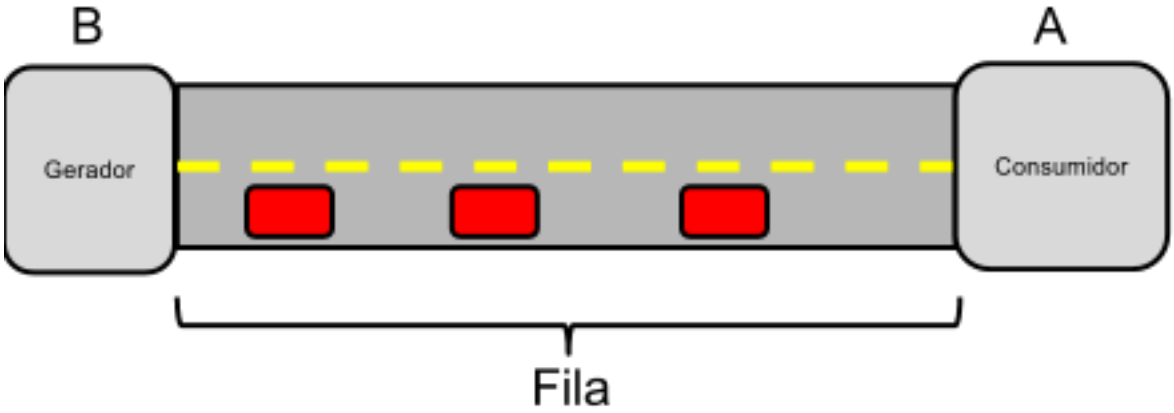

Figura 11: Modelagem de uma via urbana.

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 grande 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 11. 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, até que o consumidor A consuma eventos, liberando espaço na fila.

5.2.2 O componente Gerador

Assim como ilustrado na Figura 11, 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.

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 se justifica pela simplicidade da 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 gerar um ou mais eventos distintos

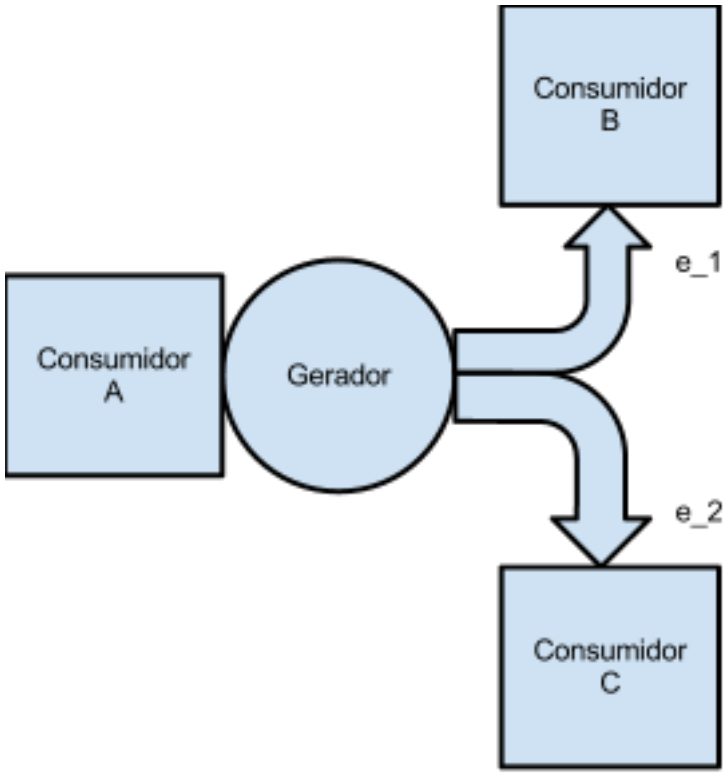

Figura 12: Um processo consumidor que gera eventos. No caso ilustrado, o consumidor A gera, através de um componente Gerador, simultaneamente os eventos e_1 e e_2 que são então encaminhados para os consumidores B e C

por esse gerador, que seriam então inseridos no sistema. Um exemplo prático desta aplicação é a simulação do fluxo de trabalho de uma companhia, onde a chegada de um documento pode disparar duas tarefas diferentes a serem feitas em paralelo (Figura 12).

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 8). 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 implementada em uma forma minimalista. Alguns detalhes do *framework* foram simplificadospara esta implementação, umas vez que se trata de uma prova de conceito.

Neste capítulo serão tratados detalhes de implementação do framework de simulação, particularidades de implementação e decisões de *design* de software, 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 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 13 (Uma versão expandida, portanto mais completa, deste diagrama encontra-se 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 de como foi implementado o *framework* é ilustrada no apêndice D. A partir dela pode-se ilustrar a modularização dos componentes do projeto e a divisão dos módulos do *framework*. Os módulos são descritos a seguir:

- **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, do subdiretório *t100*.

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 6.4.

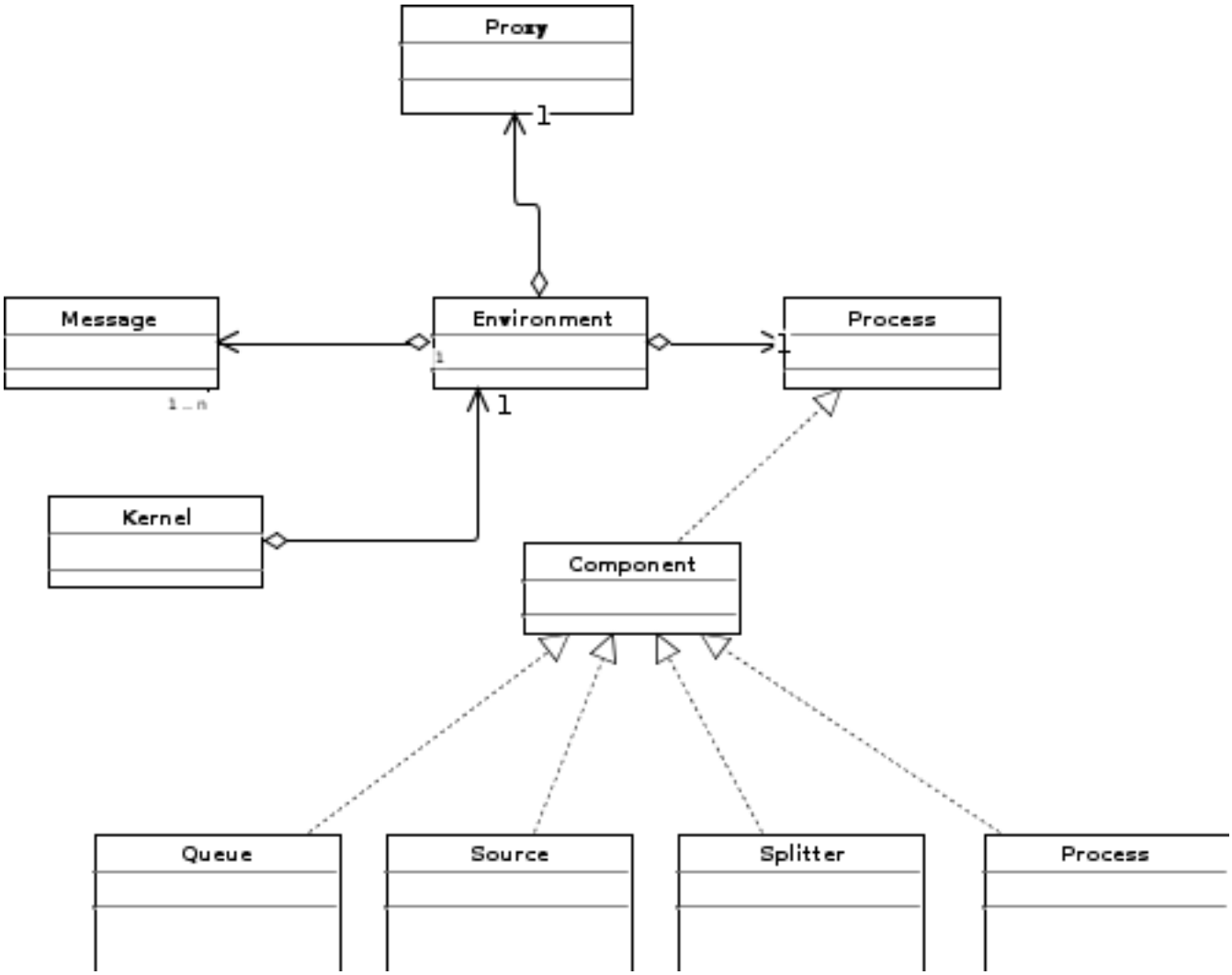

Figura 13: Diagrama simplificado de classes do *framework*.

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 figura 15, o proxy mínimo de comunicação deve oferecer os seguintes métodos: *send, receive, listen.*

Para a implementação da camada de comunicação do *framework* de simulação, foi adotada uma estratégia simplista, utilizando-se das bibliotecas de comunicação disponibilizadas pela linguagem adotada.

Toda a comunicação se faz por meio de troca de mensagens utilizando *sockets* sobre o protocolo *TCP/IP.*

6.3.1 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 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.

O objeto *Message* possui dois métodos especiais para tratar da sua serialização: o método *serialize* e o método de estático *recreate.*

O método *serialize* converte o atual objeto *Message*, com o seu devido conteúdo, em um *stream* de *bytes.* Estes *bytes* é que são transmitidos de um nó do

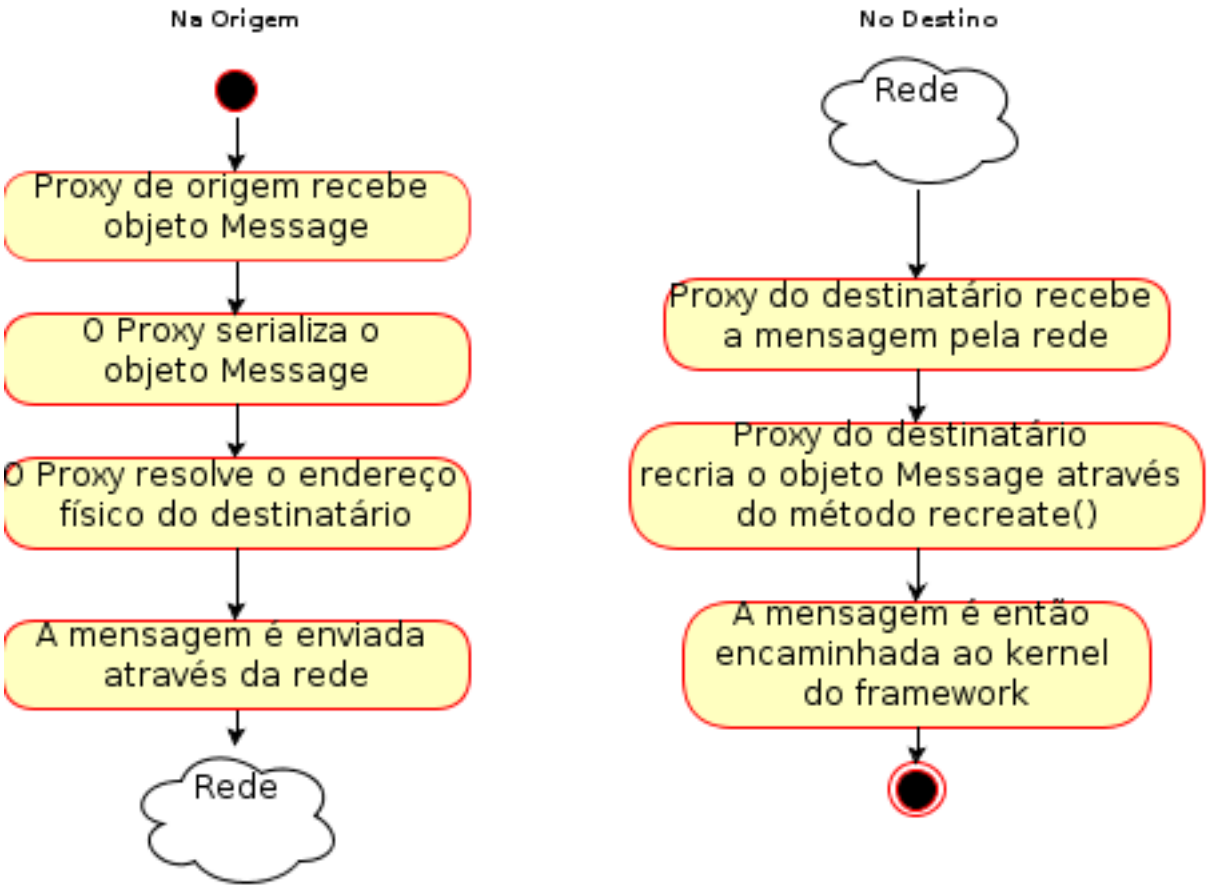

Figura 14: Diagrama de fluxo de envio de mensagem entre dois *environments* distintos.

sistema para outro quando há a necessidade de se transmitir uma mensagem.

Uma vez que a mensagem chega ao seu destino, o *proxy* local é encarregado de recriar a mensagem com base nesse *stream* de *bytes*. Para isso ele usa o método *recreate*. Este método é um método estático que instancia um novo objeto do tipo *Message*, com base no *stream* serializado recebido, e devolve essa nova instância. O fluxo do processo de envio de uma mensagem de um nó do sistema para um *environment* remoto pode ser visualizado na figura 14

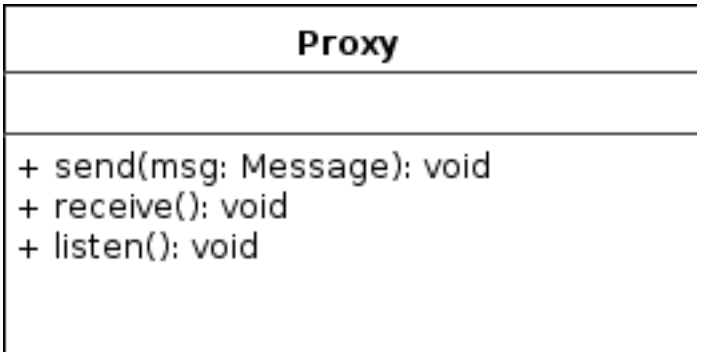

Figura 15: Diagrama de classes da classe *proxy*.

6.3.2 O método send do Proxy

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.

O mesmoo método *send* é utilizado para despachar componentes em uma migração. Mais detalhes sobre a implementação da migração de objetos é tratado na subseção 6.4.3

6.3.3 O método receive do Proxy

O método *receive* do *Proxy* do sistema é o método que é responsável por receber todas as mensagens externas direcionadas ao *environment*. Este método deve executar ininterrupdamente durante todo o processo de simulação.

Para se evitar que o método *receive* bloqueie a execução do sistema, este é executado de forma assíncrona aos demais componentes. Isto é feito utilizando a *API* de manipulação de *threads* de baixo nível da linguagem *Python*.

O código listado no apêndice E ilustra uma versão bastante simplificada da implementação do método *receive* de forma assíncrona usando a *API* de baixo

nível de *threads* da linguagem *Python*.

O construtor da classe (método **__init__**) recebe os parâmetros básicos para inicializar o *proxy* e dispara uma chamada assíncrona (**thread.start_new_thread(...)**) passando como argumento uma referência para o método que se deseja executar de forma assíncrona (neste caso, o método **__receiver__**). Com o método **__receiver__** passa a ser executado ininterrupdamente, em paralelo à função que executou a instanciação do objeto *Proxy*.

6.4 Implementando os Componentes

6.4.1 A classe base Process

6.4.2 Os componentes básicos

6.4.3 Migração de componentes

6.5 Implementando o Environment

6.6 Exemplos de aplicação

Esta seção tem como fim demonstrar a estrutura básica de um código utilizando o *framework* implementado durante este projeto. O que é apresentado são exemplos simples que visam apenas ilustrar o funcionamento do *framework* desenvolvido, assim como ilustrar o uso da biblioteca de componentes do *framework*.

6.6.1 Uma aplicação típica

O código ilustrado pela listagem 6.1 apresenta uma aplicação mínima de simulação utilizando o *framework* desenvolvido neste trabalho.

As linhas [01] e [02] importam as bibliotecas necessárias para utilizar o *fra-*

mework. A biblioteca **Simulator** é a responsável por gerenciar a simulação. Ela se encarrega de orquestrar a comunicação entre os componentes.

A biblioteca **base_components** possui os componentes básicos para a construção de uma simulação. É com esses componentes que o modelo a ser simulado deve ser construído.

As linhas [07] e [08] criam funções que são passadas para o componente que será criado. Essas funções determinam o comportamento do componente durante a simulação.

As linhas [11] e [12] criam dois componentes: uma fila de eventos e uma fonte geradora de eventos. Para a construção do gerador é passado como parâmetros as funções criadas nas linhas [07] e [08]. A expressão *creation_tax_expression* rege sobre a frequência que cada novo evento é criado pelo gerador de eventos, enquanto a expressão *execution_time_expression* representa o valor de cada evento criado. Este valor do evento é o que determina o custo do seu processamento pelo componente processador.

Um outro parâmetro passado durante a criação do componente **Source** é o parâmetro *output*. Este parâmetro determina a qual componente a saída do gerador estará ligada. No exemplo aqui discutido, a saída do gerador está ligada à fila declarada na linha [11].

Na linha número [16] é declarado o componente **Process**, e é passado como parâmetro para este uma lista de todos os componentes que possuem suas saídas atreladas à entrada do processador.

Por fim, entre as linhas [18] e [21] é criado o componente *Simulator*, responsável por gerir a simulação. Paar isto é passado ao contrutor do simulador uma lista com todos os componentes que contemplam a simulação. Com posse de uma instância do objeto *Simulator*, é requerida a sua execução chamando o método *run(...)* e passando como parâmetro o número de passos a serem simulados.

As linhas [22] e [24] obtêem dados referente a simulação (o tamanho da fila de

eventos, ao final da simulação) e devolvem para o usuário através da saída padrão.

Listing 6.1: "Exemplo de uma aplicação utilizando o *framework* desenvolvido"

```
[01] from t100.core.simulator import Simulator
[02] from t100.core.base_components import *
[03]
[04]
[05] if __name__=='__main__':
[06]
[07]     execution_time_expression = \
                                     lambda _ : 60+random.randint(-30,+30)
[08]     creation_tax_expression = lambda _ : 2.0/(60*5)
[09]
[10]     acc = 0
[11]     q = Queue()
[12]     s = Source(output=q,
[13]                creation_tax_expression=creation_tax_expression,
[14]                execution_time_expression=execution_time_expression)
[15]
[16]     p = Process(inputs=[q])
[17]
[18]     steps_number = 60*60
[19]
[20]     simul = Simulator(components=[q,s,p])
[21]     simul.run(untill=steps_number)
[22]     q_size = len(simul.components['queue'][0])
[23]
[24]     print q_size
[25]
```

6.6.2 Modelos com vários componentes

6.6.3 O log de execução da simulação

Em ambos os exemplos ilustrados até então, o resultado da simulação era dado pelo tamanho da fila de eventos, que era aferida diretamente e impressa na saída padrão. Porém, o sistema desenvolvido possui uma ferramenta que possibilita traçar todas as atividades do sistema, além de acompanhar o ciclo de vida de cada componente ou evento do sistema.

Para habilitar a ferramenta de *log* do sistema basta, ao se criar um objeto do tipo *Simulator*, habiliatra o modo *verbose*, passando um parâmetro extra **verbose=True**, além de passar como parâmetro um objeto do tipo *file_like_object*, representando o arquivo onde o *log* será armazenado.

Um exemplo de como instaciar um objeto **Simulator** para gerar um log de saída:

```
output_file = open('/tmp/simulation.log', 'w') # Cria um arquivo chamado
simul = Simulator(components=[q1,q2,s1,s2],
                  verbose=True,
                  output_file=output_file)
```

6.6.4 Escalabilidade do framework

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

APÊNDICE D – Árvore de diretórios

Listing D.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
```

APÊNDICE E – Classe com método assíncrono

Listing E.1: "Exemplo de chamada de métodos assíncronas na classe *Proxy*"

```
# Antonio Ribeiro, UNIFEI - 2012
# T100 simulation framework
# Assyncronous proxy receiver example

import socket
import thread
import time


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 i
        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()
```

Referências

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

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

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.

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.

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.