Sumário

1 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.

1.1 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.

Some-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 distri-

buída 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.

1.1.1 Encapsulamento

Uma das funções de um *framework* é o de encapsular diversos elementos que não tratam diretamente da simulação, mas sim 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. Encapsulando 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.

1.1.2 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 *fra-*

mework 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.

1.1.3 Reusabilidade

Ao se optar pelo desenvolvimento de um *framework* o responsável pelo seu *design* deve permitir que componentes de interesse sejam trocados ou mesmo customizados. Isso faz com que que o mesmo código escrito para um framework funcione mesmo depois da troca de algum componente deste framework.

No caso da simulação distribuída, componentes como protocolo de sincronização ou sistema de balanceamento de cargas poderiam ser plugáveis, o que permitiria a reutilização do mesmo código de simulação no mesmo framework, porém com caracterúisticas diferentes. Isso leva à uma reutilização de código, economizando no desenvolvimento e dinamizando a comparação entre diferentes soluções para um mesmo modelo.

1.2 Soluções Existentes

Uma proposta de *framework* para simulação distribuída foi proposta em (CRUZ, 2009), baseada em troca de mensagens suportando tanto *MPI* qaunto *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 entre os nós do sistema.

Algumas propostas como a *Remote Call Framework (RCF)*, *ClassdescMP* e diversas implementações do *MPI* e de *PVM* provém diversas 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 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.

1.3 Arquitetura proposta

A fim de suprir todas as necessidade de um sistema que suporte plenamente a migração de processos lógicos, é apresentado na 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.

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.

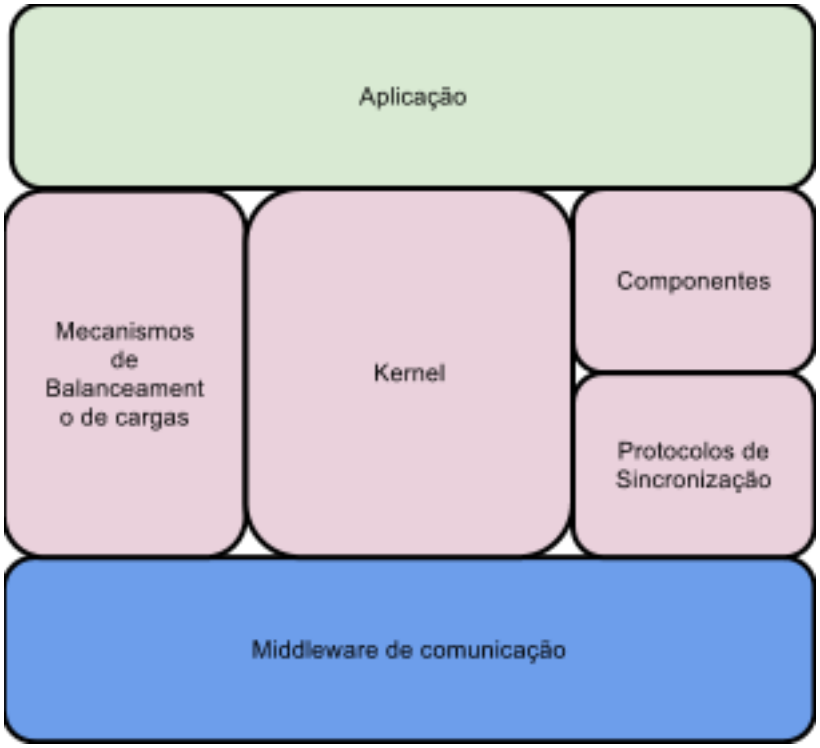

Figura 1: Camadas da arquitetura do *framework*.

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 1.2, 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.

1.4 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 no capítulo sete é demonstrado alguns exemplos de simulação utilizando o framework desenvolvido.

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

2 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.

2.1 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* (excessões à esse caso serão tratados a diante).

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

$$e_k = \{p_i | i < n\} \tag{2.1}$$

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

$$p_k \in e_l \rightarrow \nexists e_m, p_k \in e_m, e_l \neq e_m \tag{2.2}$$

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 por troca de mensagens.
- Serialização do conteúde de um processo lógico.
- Migração de um processo de um *environment* para outro.
- 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 2.4, é a união da capacidade de serialização de um processo e da comunicação entre *environments*.

2.2 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.

2.2.1 Estrutura interna

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

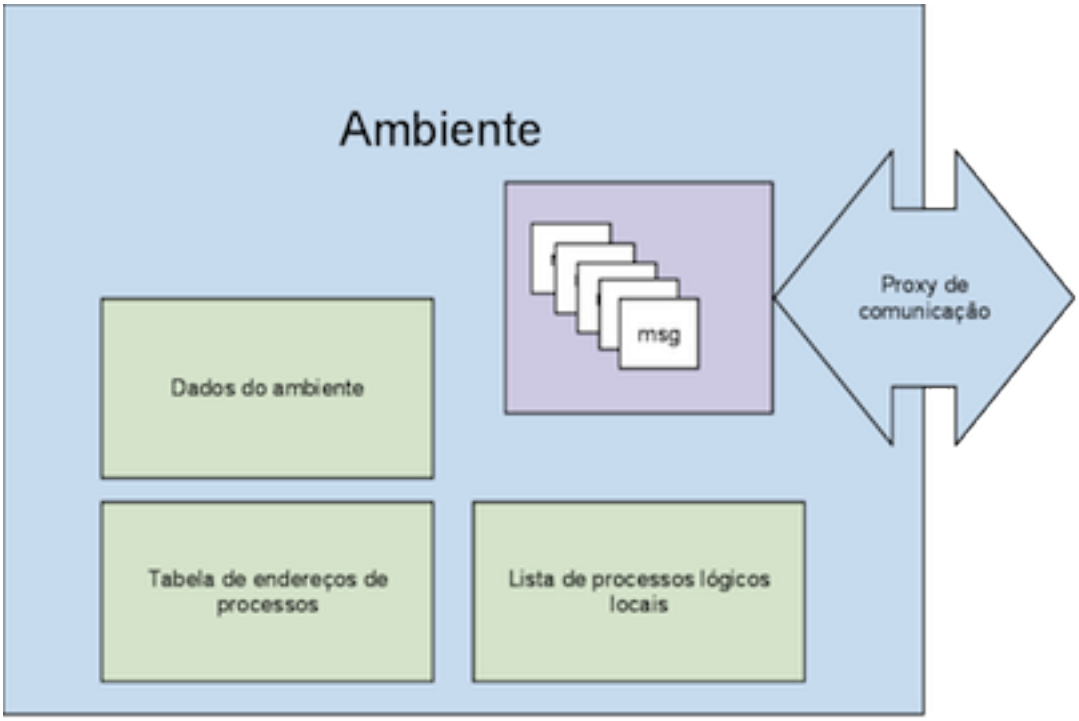

Figura 2: A arquitetura interna de um *environment*.

- 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.

Ao contrário do que será descrito na Seção 2.3 para o componente *process*, a comunicação com um *environmet* se dá somente via 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 (salvo excessões descritas na Seção 2.8)

2.2.2 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 externar serão tratadas na seção 2.5

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

2.2.3 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. A tabela de processos lógicos 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. O motivo de se utilizar o endereço lógico é que este, ao contrário do endereço físico, este 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 in;umeras 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 2.4.1.

2.3 O componente process

Um processo é a menor unidade de processamento em um sistema de simulação de eventos discretos. É ele o responsável por retirar cada evento a ser executado da fila de eventos futuros e executá-lo. A representação de um processo lógico na arquitetura aqui descrita se dá pelo componente *process*. Este deve ser capaz de proporcionar mecanismos para a implementação da simulação propriamente dita (uma área onde o usuário do *middleware* implementará a sua simulação), mecanismos de troca de mensagens, serialização e locomoção (migração) de processos..

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 do *middleware* aqui descrito, 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 traca de mensagem entre dois objetos do tipo *Process* deve ser encapsulado e resolvido pelo próprio objeto, não transferindo assim ao usuário ...

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, ou seja, é uma comunicação que foi iniciada pelo usuário do *middleware*, é feita através do endereço lógico do processo. Isto garante que o usuário do *middleware* não precise se adentrar em estruturas fícicas 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.

Por consequência, quando uma comunicação não partiu diretamente do usuário do *middleware* (como é o caso descrito anteriormente. Após a mensagem chegar no *environment* este agora vai encaminhá-la ao destinatário. Este encaminhamento é feito automaticamente pelo sistema, não necessitando de demais intervenções do usuário do *middleware*.), neste caso o endereço usado para se comunicar com o processo é o seu endereço físico na rede.

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.

2.3.1 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 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. 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 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

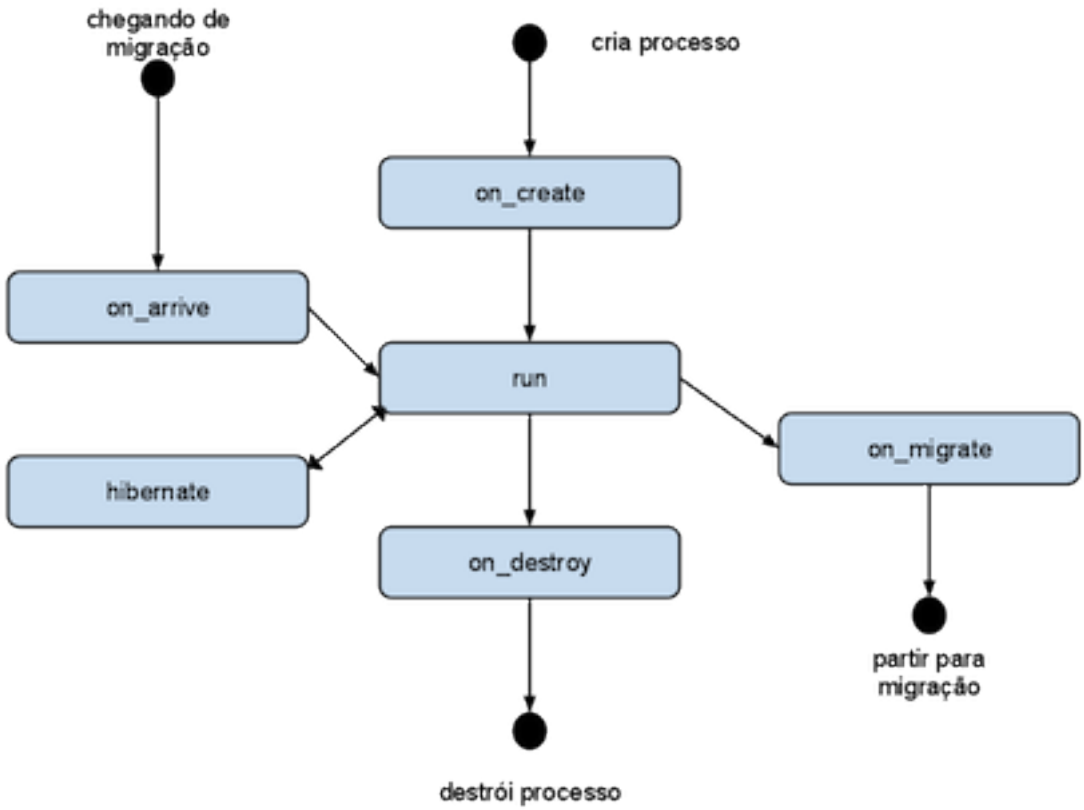

Figura 3: Ciclo de vida de um processo lógico.

mensagens externas, migrações, etc.

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.

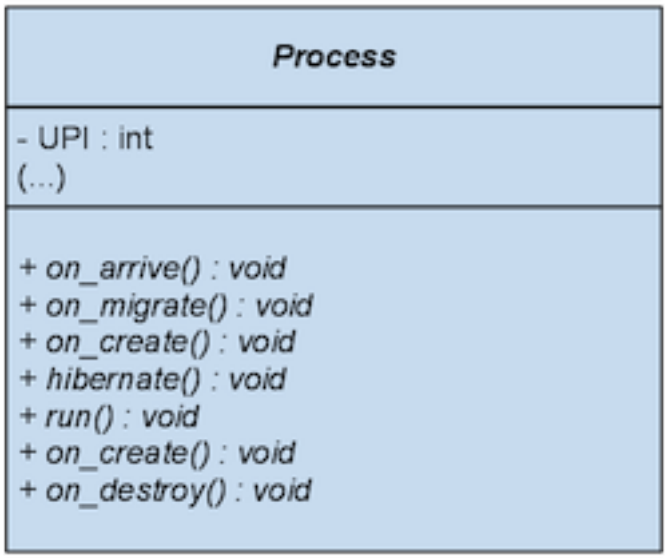

Figura 4: Representação em diagrama de classe do componente *process*.

2.3.2 Serialização de um processo

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.

2.4 Migração de processos

Um processo pode migrar de um ambiente para outro quando necessário ou conveniente para a simulação. Quando a migração ocorrer, uma série de ações ocorrem na seguinte ordem:

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.

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.

2.4.1 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 a 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, al;em 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.

2.5 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.

2.5.1 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 enviala 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).

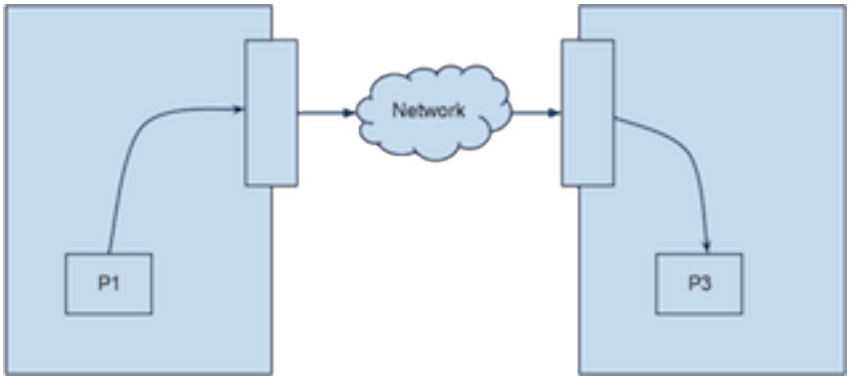

Figura 5: Comunicação indireta *proxy-process*.

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

2.5.2 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

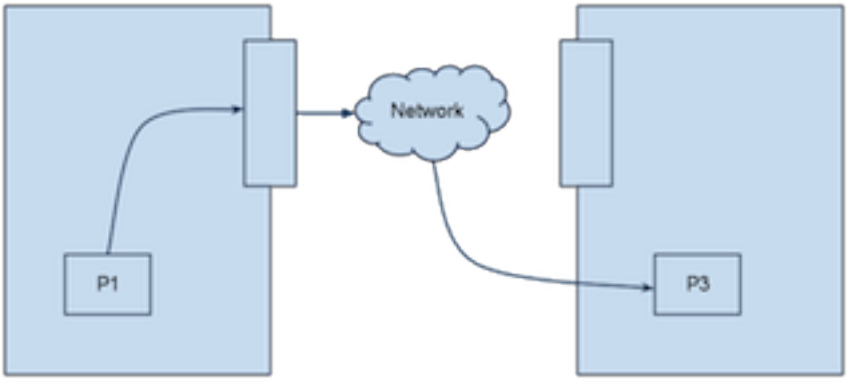

Figura 6: Comunicação direta *proxy-process*.

Figura 7: Comunicação direta *process-process*.

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 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.

2.6 Serialização em grande escala

2.7 Comunicação grupal

2.8 Migraçoes em grande escala e migrações de ambientes

3 Arquitetura do framework de simulação

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.