PAWEB - Uma plataforma para desenvolvimento de aplicativos web utilizando o modelo de atores

Bruno Takahashi Carvalhas de Oliveira

Dissertação apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para obtenção do
título de
Mestre em Ciência da Computação

Programa: Mestrado em Ciência da Computação

Orientador: Prof. Dr. Francisco Reverbel

Durante o desenvolvimento deste trabalho o autor recebeu auxílio financeiro da CAPES

São Paulo, agosto de 2012

PAWEB - Uma plataforma para desenvolvimento de aplicativos web utilizando o modelo de atores

Esta tese/dissertação trata-se da versão original
do aluno (Bruno Takahashi Carvalhas de Oliveira).

Agradecimentos

Gostaria de prestar meus mais sinceros agradecimentos às seguintes pessoas e entidades:

- aos meus pais (Rui Oliveira e Celina Oliveira) e à minha noiva (Andrea Cornaglia), pelo apoio incondicional que me deram;
- ao meu orientador, Prof. Dr. Francisco Reverbel, por toda sua paciência e dedicação;
- ao Google, empresa onde trabalho, por fornecer a flexibilidade necessária para que eu pudesse concluir esta dissertação;
- à CAPES, pelo apoio financeiro que foi fundamental no período inicial da elaboração deste trabalho.

Resumo

Existem várias linguagens e plataformas que permitem a programação baseada no modelo de atores, uma solução elegante para a programação concorrente proposta há algumas décadas. Segundo esse modelo, implementa-se o programa na forma de uma série de agentes que são executados em paralelo e se comunicam entre si somente por meio da troca de mensagens, sem a necessidade de memória compartilhada ou estruturas tradicionais de sincronização como semáforos e *mutexes*. Uma das áreas nas quais esse modelo seria particularmente adequado é a programação de *aplicações web*, isto é, aplicações cujas lógicas de negócios e de dados residem num servidor e que são acessadas pelo usuário por intermédio de um *navegador*. Porém, existem muitos obstáculos ao desenvolvimento de aplicações desse tipo, entre eles a falta de linguagens e ferramentas que permitam integrar tanto o servidor quanto o cliente (navegador) no modelo de atores, as dificuldades de conversões de dados que se fazem necessárias quando o servidor e o cliente são desenvolvidos em linguagens diferentes, e a necessidade de contornar as dificuldades inerentes aos detalhes do protocolo de comunicação (HTTP). O PAWEB é uma proposta de uma plataforma para o desenvolvimento e execução de aplicações *web* que fornece a infraestrutura necessária para que tanto o lado cliente quanto o lado servidor do aplicativo hospedado possam ser escritos numa mesma linguagem (Python), e possam criar e gerenciar atores que trocam mensagens entre si, tanto local quanto remotamente, de maneira transparente e sem a necessidade de implementar conversões de dados ou outros detalhes de baixo nível.

Palavras-chave: modelo de atores, programação concorrente, Python, sistemas distribuídos, *web*

Abstract

There are several programming languages and platforms that allow the development of systems based on the actor model, an elegant solution for concurrent programming proposed a few decades ago. According to this model, the program is implemented in the form of several agents that run concurrently and only communicate amongst themselves through the exchange of messages, without the need for shared memory or traditional synchronization structures such as semaphores and mutexes. One of the areas where this model would be particularly appropriate would be the development of web applications, that is, applications whose business and database logic reside on the server and are accessed by the user by means of a web browser. However, there are several obstacles to the development of this type of application, amongst them the lack of languages and tools that allow for the integration of both the server and the client (browser) into the actor model, data conversion difficulties arising from using different programming languages on the server and the client, and the need to circumvent the inherent difficulties posed by the details of the communications protocol (HTTP). PAWEB is a proposal for an application development and execution platform that supplies the infrastructure needed so that both the server and client sides of the hosted application can be written in the same language (Python) and so that they may create and manage actors that exchange messages with one another, both locally and remotely, in a transparent manner and without the need to implement data conversions or other low-level mechanisms.

Keywords: web applications, actor model, concurrent programming, distributed systems, Python, web

Sumário

Lista de Figuras

Capítulo 1

Introdução

1.1 Motivação e Propósito

O modelo de atores, uma solução elegante para concorrência proposta há algumas décadas, consiste em organizar o programa na forma de uma série de agentes que são executados em paralelo e se comunicam entre si somente por meio de trocas de mensagens [HBS73]. Nesse modelo, não há utilização de memória compartilhada, semáforos, *mutexes* e outras estruturas tradicionais de sincronização. Há linguagens, como por exemplo Erlang [Eri], que se especializam em implementar e encorajar o uso desse modelo, e há também outras linguagens nas quais o modelo de atores não é um aspecto central mas pode ser utilizado por meio de bibliotecas. Há várias linguagens e plataformas que implementam o modelo de atores distribuído, isto é, no qual os atores podem residir em máquinas diferentes dentro de uma rede e se comunicar entre si de maneira transparente.

Porém, observamos que uma das limitações é que nenhuma dessas plataformas permite a programação de aplicativos *web* com integração total do componente cliente (que é executado no navegador do usuário) ao modelo de atores. Ou seja, ainda que as mensagens possam ser trocadas dentro de uma rede, geralmente não é possível trocar mensagens com um componente cliente em execução num navegador como o Mozilla Firefox. Além disso, para realizar a comunicação entre o cliente e o servidor, normalmente é necessário recorrer à definição e à implementação de um protocolo de baixo nível utilizando modelos como o Ajax, bem como utilizar a seriação de objetos (formato JSON ou XML, por exemplo) para permitir o intercâmbio de dados estruturados. Embora existam várias bibliotecas e programas que visam simplificar a implementação desse tipo de solução, esses componentes geralmente não fornecem

uma maneira de transpôr o obstáculo da fronteira entre linguagens e das diferenças nos formatos de objetos, nem permitem a troca de mensagens de uma maneira simples e direta. Tais fatos, somados à característica do Javascript de ser *single-threaded*, prejudicam bastante a implementação correta de um modelo de atores que inclua o componente cliente.

O PAWEB é uma proposta de uma plataforma de desenvolvimento que permite programar tanto o servidor quanto o cliente da aplicação *web* numa mesma linguagem (Python) utilizando o modelo de atores, no qual o aspecto central é que o componente cliente (navegador) poderá executar atores da mesma forma que o servidor. Isto é feito sem que haja necessidade de qualquer preocupação especial por parte do programador para realizar o intercâmbio de dados, pois a plataforma cuidará dos detalhes necessários para que a troca de mensagens entre os atores residentes no cliente e no servidor seja transparente.

Para isso, implementaremos um *plugin* no navegador do cliente que gerenciará a máquina virtual Python e uma biblioteca que implementaremos tanto do lado cliente quanto do servidor, que possibilitará a troca de mensagens transparente entre quaisquer atores seja qual for a sua localização relativa. Isto significa programar um protocolo sobre o HTTP [FGM+99] para que os atores do plugin possam trocar mensagens com os atores do servidor, bem como elaborar estratégias para realizar a comunicação inversa (do servidor para o cliente), que é um desafio devido ao fato de que o HTTP não permite que o servidor inicie espontaneamente uma conexão com o cliente. Porém, todas as dificuldades são superáveis com a aplicação de técnicas já bastante conhecidas na programação de aplicativos web tradicionais, bastando adaptá-las ao nosso modelo, como explicaremos em maior detalhe ao longo do presente texto.

1.2 O Modelo de Atores

1.2.1 Contexto Histórico e Motivação

Desde os primórdios da computação, a execução concorrente de programas sempre foi um assunto de grande interesse teórico e prático. Basta lembrar que, antes da popularização do conceito de computador pessoal, os computadores eram equipamentos extremamente caros e portanto esperava-se que um único computador servisse a uma grande quantidade de pessoas simultaneamente. Por outro lado, os computadores pessoais durante muito tempo ignoraram a necessidade do multiprocessamento efetivo, sobrevivendo durante décadas com algumas estruturas rudimentares como programas “residentes” e rotinas

de interrupção, que serviam tanto para uma implementação rudimentar de paralelismo como para chamadas ao sistema operacional [BK93]. Porém, já no final do século passado, a capacidade de executar vários programas concorrentemente passou a ser uma característica indispensável a um computador pessoal moderno.

Originalmente, os sistemas Unix somente forneciam suporte direto para a programação concorrente por meio de *processos*, isto é, unidades de código que geralmente não compartilhavam memória entre si e só se comunicavam por intermédio de descritores de arquivos. Como explica Tanenbaum [TW97, 47], o propósito dessa concorrência era permitir que vários programas diferentes pertencentes a vários usuários diferentes fossem executados no mesmo processador em fatias distintas de tempo escalonadas pelo sistema operacional, criando a ilusão de que os processos estavam sendo executados em paralelo, também chamada "pseudo-paralelismo." Mesmo em máquinas com vários processadores o pseudo-paralelismo ainda é útil para os casos em que há mais processos sendo executados do que processadores físicos.

Embora o propósito original dos processos fosse executar vários programas diferentes, o multiprocessamento (simulado ou não) também é muito conveniente para organizar uma mesma aplicação em várias tarefas paralelas. Este modelo é especialmente útil quando se deseja obter mais desempenho num sistema com múltiplos processadores, pois se a aplicação for escrita como um único processo, somente um processador poderá executá-la de cada vez. Organizando-a na forma de vários processos que se comunicam, é possível utilizar mais eficientemente os processadores disponíveis. Porém, o fato de que os canais de comunicação entre os processos são razoavelmente escassos e de baixo desempenho, além de necessitarem da mediação constante do sistema operacional para realizar transferências de dados, levou ao surgimento do modelo de *threads*. Esse modelo introduz a possibilidade da existência de múltiplas linhas de execução simultâneas dentro do mesmo espaço de endereçamento [TW97, 54], ou seja, com compartilhamento da memória.

Porém, a programação concorrente baseada nesse compartilhamento de memória, embora sua implementação na maioria dos sistemas operacionais e plataformas seja razoavelmente direta e eficiente, apresenta uma série de desafios à programação. A necessidade de sincronização entre as *threads* no acesso a memória e outros recursos exige um cuidado especial do programador para que os dados não se tornem inconsistentes, cuidado esse que não é trivial nem mesmo num sistema de pequeno porte, e torna-se um enorme desafio num programa

maior.

A facilidade com que erros de programação podem ser introduzidos, aliada à grande dificuldade de depuração e previsão dos resultados de um programa, faz com que a programação com *threads* seja vista como uma área extremamente delicada e problemática da computação. Frequentemente a construção de sistemas grandes baseados diretamente em *threads* é desencorajada em favor de outras estruturas de concorrência que possuem um maior nível de abstração. A necessidade de tais estruturas e paradigmas mais elaborados explica o grande volume de trabalhos que lidam exclusivamente com primitivas, práticas ou arquiteturas destinadas a tornar a programação concorrente mais organizada e menos sujeita a erros. Por exemplo, a estrutura de sincronização conhecida como "monitor," introduzida por Hoare [Hoa74], foi proposta para tornar mais fácil e intuitivo o controle de acesso a dados compartilhados dentro de módulos chamados a partir de várias *threads*, ocultando do programador a utilização direta de semáforos ou *mutexes*.

Um dos modelos que se propõem a tornar a programação concorrente mais fácil e intuitiva é o modelo de atores. Originalmente proposto por Hewitt, Bishop e Steiger [HBS73] em 1973, consistia num modelo de programação conceitualmente baseado num único tipo de objeto: atores, que também podem ser considerados como "processadores virtuais" ou "contextos de execução" que executam código independentemente uns dos outros. Além da facilidade da programação, Clinger [Cli81] aponta um outro fator que motivou a criação do modelo de atores: no começo da década de 1980, já havia o prospecto do surgimento de máquinas altamente paralelas com centenas ou mesmo milhares de microprocessadores independentes, cada um com sua própria memória. E, com efeito, nos dias de hoje vemos uma tendência cada vez maior da indústria em fabricar processores com múltiplos núcleos, em grande parte devido ao fato de que a velocidade dos processadores atingiu um limite técnico que, ao que tudo indica, só poderá ser transposto com uma drástica mudança na forma como construímos processadores. Enquanto tal mudança não ocorre, a única alternativa para aumentar o poder de processamento das máquinas é utilizar o processamento paralelo. Portanto, o modelo de atores e outros trabalhos relacionados à programação concorrente e paralela vêm ganhando atenção renovada da indústria e da comunidade acadêmica nos últimos anos.

O trabalho original de Hewitt [HBS73] não fornece uma implementação concreta do modelo de atores, mas sim, como explica o autor, um formalismo abstrato que pode ser implementado de diversas formas e em diversas plataformas,

inclusive diretamente em circuitos eletrônicos. Ainda segundo Hewitt [HBS73], uma das principais vantagens dessa arquitetura é que ela prescinde completamente de estruturas de sincronização como interrupções e semáforos, pois o modelo implementa a mesma funcionalidade dessas estruturas por meio de mecanismos de alto nível.

Houve vários trabalhos dedicados à formalização do conceito de atores e a avaliação de suas das consequências lógicas, bem como a elaboração de regras e "leis" que permitem provar a corretude de um programa formulado segundo o modelo. Um exemplo é o trabalho de Hewitt e Baker [HB77]. Esse trabalho estabelece alguns axiomas e investiga como as trocas de mensagens entre os atores, consideradas como relações causais, podem ser utilizadas para construir uma base lógica para a análise formal desses sistemas. De fato, como explica Agha [Agh86, 14], há muitas vantagens em utilizar esse tipo de lógica baseada em causalidade ao invés de lidar com o conceito de tempo sob presunção de um relógio global único. Isto ocorre porque a interpretação causal de cada mensagem possui muito mais significado e utilidade do que a informação do momento preciso em que cada uma foi enviada ou da sua ordenação cronológica [Agh86, 14].

Uma outra preocupação teórica inicial foi investigar se haveria uma equivalência de expressividade entre o modelo de atores e o modelo de programação sequencial. Porém, foi demonstrado que essa equivalência não existe. Ao invés disso, como explica Agha [Agh86, 13], os atores são um agente computacional mais poderoso e geral do que processos sequenciais ou sistemas funcionais de transformação de valores, pois é possível expressar qualquer sistema funcional ou sequencial em termos de atores, mas o inverso não é necessariamente verdadeiro.

É cabível observar também que um modelo de programação bastante difundido na atualidade, por causa das interfaces gráficas com o usuário, é o modelo baseado em eventos. Nesse modelo, o programa essencialmente consiste em trechos de código que reagem a certos eventos que podem ocorrer no sistema. Como explicam Haller e Odersky [HO07], tanto a programação baseada em *threads* quanto a programação baseada em eventos podem ser unificadas sob uma única abstração com atores. Essa abstração não é somente teórica mas também prática, como podemos observar do pacote de interface gráfica `gs` da linguagem Erlang [Eri], que completamente substitui a noção de eventos pela utilização de atores para representar e interagir com os componentes gráficos.

1.2.2 Fundamentos do Modelo de Atores

Um ator é um agente computacional cujos aspectos fundamentais, na arquitetura proposta por Hewitt, Bishop e Steiger [HBS73], são a execução independente e a comunicação por meio da troca de mensagens, sem compartilhamento de memória. Isto é, um sistema elaborado segundo o modelo de atores essencialmente consiste num certo número de atores send executados concorrentemente e trocando mensagens entre si. O modelo é puramente teórico, logo não há exigências concretas sobre a infraestrutura de execução dos atores, isto é, não há suposições ou exigências sobre a utilização de processos, *threads*, ou outras abordagens. De fato, não há nem sequer a exigência da concorrência estrita. Ou seja, é possível implementar o modelo de atores num sistema com um único processador físico se houver o "pseudo-paralelismo," uma infraestrutura de alternância de contexto mediada pelo sistema operacional. O pseudo-paralelismo cria a ilusão de que vários processos estão sendo executados em paralelo [TW97, 47]. Desta forma, a escolha do suporte, isto é, a implementação concreta da infraestrutura de execução dos atores, é algo deixado inteiramente a cargo de cada linguagem ou plataforma em particular.

Um aspecto interessante é que não há nenhum impedimento técnico à possibilidade dos atores de um mesmo sistema estarem localizados em máquinas diferentes dentro de uma rede. De fato, esse tipo de organização é muito facilitada pelo fato de que não há compartilhamento de memória. Portanto, basta a plataforma de execução fornecer um mecanismo para a localização de outros atores na rede e um protocolo para trocas de mensagens. Este modelo é implementado por algumas linguagens e bibliotecas, como por exemplo Erlang [Eri], Stage [AE09] e SALSA [VA01].

Agha [Agh86, 12] explica que, do ponto de vista formal, um ator é um agente computacional que mapeia cada mensagem recebida a uma tupla de três elementos que consiste em:

1. um conjunto de mensagens enviadas a outros atores;
2. uma nova tupla de três elementos que define um novo comportamento (que governará a resposta do ator à próxima mensagem); e
3. um conjunto de novos atores criados.

Isto significa que a atividade de um ator consiste essencialmente em receber mensagens, processá-las, enviar mensagens, criar novos atores e mudar seu próprio comportamento. Ainda segundo Agha [Agh86, 13], a possibilidade da

mudança de comportamento a cada mensagem recebida permite que os atores sejam *sensíveis ao histórico*. Em outras palavras, o ator pode manter um estado interno que vai se modificando à medida que mensagens são recebidas. Esse estado interno ou *sensibilidade ao histórico* fundamentalmente diferencia um ator de uma aplicação de uma função matemática para transformação de valores.

Armstrong [Arm07, 137] explica o modelo de concorrência da linguagem Erlang [Eri], que implementa o modelo de atores de maneira bastante fiel. Segundo o autor, esse modelo de concorrência emula de uma forma intuitiva a interação entre seres humanos no mundo real, pois cada um de nós é essencialmente uma unidade autônoma de pensamento e nossa memória é privada e inacessível aos outros, de forma que só podemos nos comunicar uns com os outros por meio de mensagens.

No modelo formal, não há restrições à comunicação entre quaisquer dois atores de um sistema. Porém, em sistemas concretos, geralmente existe a exigência prática de que haja uma maneira de especificar, de maneira inambígua, qual é o ator destinatário de uma mensagem. Para isso, geralmente se utiliza o conceito de *endereço de ator*. Um endereço é essencialmente um valor, cujo tipo é dependente da implementação específica em questão, que permite à plataforma localizar o ator a que se destina a mensagem. Esta exigência gera uma restrição adicional bastante relevante: um ator só pode se comunicar com os atores cujos endereços ele possui (pressupõe-se que o ator conheça o próprio endereço, ou que haja na linguagem uma primitiva ou função que forneça tal endereço). Em determinados sistemas, é possível "sintetizar" novos endereços a partir de outras informações, enquanto que em outros o endereço de um ator é algo opaco que só pode ser fornecido pelo próprio ator ou por algum outro ator que já o conheça anteriormente. Desta forma, aparece o conceito de "localidade," isto é, um ator só pode se comunicar com atores que sejam de seu conhecimento prévio, e não tem acesso e nem pode interagir com demais atores do sistema.

Como explica Agha [Agh86, 150], o modelo de atores também é muito conveniente para implementar o encapsulamento e a composição de funcionalidades, permitindo assim a construção de sistemas complexos. Agha explica que o encapsulamento é obtido por meio da utilização de atores *recepcionistas*, método esse que consiste em designar, num componente, um ator cuja finalidade é trocar mensagens com o "exterior" (i.e. atores que não fazem parte daquele componente) e traduzir ou distribuir essas mensagens para os atores

internos. O ator recepcionista também pode realizar a tarefa contrária, isto é, encaminhar as respostas dos atores internos aos atores que as requisitaram.

Este tipo de organização, aliado ao conceito de "localidade" explicado anteriormente, permite o encapsulamento completo de componentes do sistema, uma vez que é impossível, para um ator externo ao componente, obter o endereço de qualquer ator do componente exceto o "recepcionista." Desta forma, a organização e a implementação dos atores do componente podem ser alterados livremente, bastando manter intacto o protocolo do ator "recepcionista."

Uma consequência interessante e bastante conhecida da combinação desse mecanismo com a capacidade dos atores de mudar de comportamento dinamicamente é a possibilidade de substituir um componente inteiro do programa em tempo de execução, sem a necessidade de parar o sistema. Este é o exemplo utilizado por Armstrong no caso de sistemas de telefonia, que necessitam permanecer em operação ininterrupta mesmo durante atualizações [Arm03]. Para realizar isso, basta o componente substituir seus atores por novos atores com novos comportamentos (uma versão atualizada do código, por exemplo) e manter inalterada a interface do ator "recepcionista." Naturalmente algum cuidado deve ser tomado para que o recepcionista retenha as requisições numa fila de espera durante o processo de atualização e só continue a repassá-las uma vez que a atualização esteja completa e consistente, mas essa é uma tarefa administrativa razoavelmente simples. A grande vantagem dessa abordagem é que o restante do sistema não é influenciado pela atualização, pois o componente não muda de endereço nem de interface, e nem sequer chega a ficar indisponível. No máximo, a retenção temporária das requisições durante a atualização será vista pelos demais componentes como uma lentidão ou congestionamento transitório sem maiores consequências.

A composição de atores, também explicada por Agha [Agh86, 150] também é obtida de maneira relativamente simples. Nesse caso, um ator mantém o endereço de um outro ator para o qual encaminha as mensagens enviadas e recebidas, tornando-se, como no caso do encapsulamento, uma espécie de "recepcionista" do ator. A diferença é que nesse caso o ator aplica seu próprio processamento antes de enviar a mensagem ao ator contido ou após receber sua resposta.

1.2.3 Linguagens com Suporte ao Modelo de Atores

Como já mencionamos anteriormente, o modelo de atores é um modelo lógico independente de implementação. Logo, em teoria, o modelo pode ser

implementado em qualquer linguagem que fornece as primitivas necessárias. Particularmente, basta que haja suporte a multiprocessamento de algum tipo (processos, *threads*, ou alguma forma de emulá-los) e primitivas elementares de sincronização para possibilitar a implementação de uma infraestrutura de troca de mensagens.

Porém, embora haja viabilidade teórica da implementação de um modelo de atores em quase todas as linguagens e plataformas, existem certas linguagens que já fornecem esse tipo de suporte embutido como parte de sua própria estrutura, muitas vezes com construções sintáticas já adaptadas para este propósito. As linguagens assim concebidas costumam apresentar uma expressividade muito melhor do que linguagens que dependem de bibliotecas ou componentes externos para implementar o modelo de atores. Um programa escrito seguindo o modelo de atores tipicamente será muito mais claro e fácil de manter se for implementado numa linguagem que já fornece suporte "nativo" a atores e trocas de mensagens. Infelizmente, ainda não há uma grande aceitação de linguagens desse tipo por parte da indústria, razão pela qual a opção de adaptar uma linguagem existente utilizando uma biblioteca pode ser uma concessão necessária do ponto de vista prático, como é o caso do PAWEB. Examinaremos a seguir alguns exemplos de linguagens dos dois tipos, isto é, aquelas concebidas desde o início para fornecer suporte ao modelo de atores e aquelas que não têm atores como ponto central de seu funcionamento, mas para as quais existe uma biblioteca ou modificação que lhes permite fornecer esse suporte.

Em meados da década de 1980, como menciona Agha [Agh86, 1], várias linguagens de programação como o SIMULA-67 [BDMN73], SmallTalk [GR83] e o CLU [Lis75], já refletiam uma mudança da ênfase de procedimentos que agem sobre dados passivos (modelo procedural) em direção a mensagens que ativamente processam os dados. Algumas linguagens dessa época já forneciam estruturas similares ao modelo de atores, como por exemplo a estrutura de sincronização conhecida como "monitor," introduzida por Hoare [Hoa74].

Não podemos deixar de mencionar a linguagem Act, introduzida em 1981 por Lieberman [Lie81]. Como explica o autor, um dos propósitos da linguagem era justamente abrandar a barreira clássica entre "dados" e "procedimentos" ao empregar objetos ativos que ao mesmo tempo continham dados e eram capazes de agir por si próprios. Assim, o Act foi uma das primeiras linguagens a adotar os preceitos do modelo de atores como filosofia central de projeto. No final daquela década, Athas e Boden [AB88] introduziram a linguagem Cantor, voltada à computação científica, também baseada no modelo de atores.

Porém, talvez um dos exemplos mais representativos de uma linguagem baseada no modelo de atores é a linguagem Erlang [Eri]. Como explica Armstrong [Arm03], o criador dessa linguagem, o propósito original do Erlang era ser uma linguagem para a programação de sistemas tolerantes à falha, especialmente em ambientes em que a disponibilidade ininterrupta do sistema era necessária, como por exemplo num sistema telefônico.

Como explica aquele autor, a concorrência era um dos requisitos fundamentais na elaboração da plataforma. Afinal, uma vez que dezenas de milhares de pessoas utilizam o sistema simultaneamente, a plataforma deveria ser capaz de lidar com milhares de tarefas simultâneas. Outro requisito apontado por Armstrong era a capacidade de distribuição, isto é, o sistema deveria estar estruturado de forma a permitir a possibilidade de utilizar várias máquinas para distribuir as tarefas, sem que isso acarretasse uma grande dificuldade adicional de programação.

Finalmente, Armstrong também observa que os sistemas que se pretendia programar com a linguagem eram tipicamente sistemas grandes (milhões de linhas de código), então a linguagem e plataforma deveriam fornecer suporte para uma boa organização do código e modelagem das interações. Além disso, era necessário que o sistema permanecesse em operação contínua, mesmo durante atualizações de componentes, e também que o sistema fosse tolerante a falhas.

Como bem resumiu Agha [Agh86], o modelo de atores é bastante apropriado para atender a esses requisitos. Afinal, os atores são inerentemente concorrentes, a comunicação por mensagens se adapta bem à implementação de atores distribuídos numa rede, o encapsulamento e composição de atores permite desenvolver e manter um sistema grande de maneira organizada. Além disso, a possibilidade dos atores mudarem de comportamento em tempo de execução torna possível a substituição da funcionalidade de componentes inteiros do sistema sem interrompê-lo. A tolerância a falhas é obtida organizando-se os atores numa hierarquia na qual os erros num determinado sistema são reportados para atores hierarquicamente superiores que podem tomar ações corretivas sem que o restante do sistema seja afetado [Arm03].

A linguagem Erlang [Eri] se destaca não somente pela sua implementação do modelo de atores mas também pela sintaxe simples utilizada para criar, gerenciar e trocar mensagens entre atores. Como explica Armstrong [Arm07], algumas poucas palavras-chave e operadores são suficientes para realizar essas tarefa, particularmente `spawn`, o operador `!` e a primitiva `receive..end`.

Desta forma, se bem escrito, o código de um programa baseado em atores em Erlang tende a ficar bastante claro e sintaticamente elegante.

A máquina virtual Erlang é capaz de organizar os atores de maneira a distribuí-los entre os processadores disponíveis, resultando, nas palavras de Ayres et al. [AE09], num modelo impressionante de concorrência em espaço de usuário que dá suporte a rápida criação e alternância entre processos e com excelentes resultados.

Podemos citar também, como outro exemplo de linguagem moderna baseada no modelo de atores, a linguagem SALSA introduzida por Varela e Agha [VA01]. Segundo explicam esses autores, a motivação principal da linguagem foi a necessidade de construir aplicativos que possam se adaptar facilmente a mudanças em seu ambiente em tempo de execução, como por exemplo a reconfiguração ou migração de módulos ou sub-componentes para outras máquinas sem interromper a execução do sistema. Uma diferença entre SALSA e Erlang é que não há uma máquina virtual SALSA; ao invés disso, o pré-processador elaborado por Varela e Agha [VA01] transforma o código SALSA em código Java, utilizando portanto o compilador e a máquina virtual Java.

Um outro exemplo importante de linguagem que dá suporte ao modelo de atores é a linguagem Scala [O+03], que foi introduzida por Odersky [OAC+04] e atualmente conta com uma grande popularidade na indústria. Diferentemente da linguagem Erlang, a linguagem Scala não é inteiramente focada no modelo de atores, oferecendo ao invés disso uma série de ferramentas que permite a programação multiparadigma, entre elas o suporte ao paradigma orientado a objetos e o funcional. De fato, como explica Odersky [OSV08], uma das filosofias que norteou a elaboração da linguagem foi a "escalabilidade," isto é, a possibilidade de estender a linguagem com módulos e bibliotecas, que podem introduzir uma nova sintaxe por meio da adaptação de primitivas sintáticas pré-existentes. Desta forma, a linguagem fornece um módulo que implementa o modelo de atores utilizando a mesma filosofia da linguagem Erlang [Eri], e com uma sintaxe também inspirada nessa linguagem. A linguagem Scala não possui máquina virtual. Ao invés disso, o compilador gera código intermediário Java que é executado na máquina virtual Java. Este método pode ser comparado com o utilizado pela linguagem SALSA, que, como explicam Varela et al. [VA01], traduz o código para Java e utiliza a máquina virtual Java para execução. Porém, como explica Odersky [OSV08], o compilador Scala é bastante poderoso e, para permitir as estruturas ricas e flexíveis disponíveis em Scala que não possuem correspondentes em Java, muitas vezes explora a

funcionalidade da máquina virtual de formas não convencionais. Por exemplo, compilador faz uso de exceções internas para o controle normal do fluxo de execução de certas primitivas sintáticas.

1.3 Arquitetura de Complementos de Navegador

1.3.1 Conceito de Complemento

Um *plugin* ou *complemento* é um componente, tipicamente fornecido por um terceiro, que se integra com um programa existente, estendendo ou modificando sua funcionalidade. Os complementos são uma forma bastante conhecida de alterar ou estender programas, e existem desde a década de 1970. Talvez um dos primeiros exemplos de um programa que fornecia suporte ao conceito de complemento era o editor de textos EDT. Como explica sua documentação [Dig83], o editor permitia ao usuário fornecer programas externos para operar sobre o *buffer* de texto sendo editado, de forma a implementar assim operações de texto que não eram originalmente fornecidas pelo editor.

Ao longo do tempo, muitos outros programas adotaram o conceito de complemento, mas um dos exemplos mais notórios da atualidade são os navegadores. Um dos primeiros navegadores amplamente utilizados foi o Netscape Navigator [Net], que, desde suas versões mais antigas, já dava suporte o conceito de *plugin*. Como explica Hoff [Hof99], os *plugins* do Netscape são uma maneira de permitir que o navegador apresente formatos de conteúdo que nem sequer existiam na época em que o navegador foi criado.

O navegador Firefox é derivado do navegador Mozilla, que, por sua vez, é derivado do navegador Netscape. Ao longo dessas mudanças, o sistema de *plugins* foi preservado numa forma relativamente próxima à sua forma original. Particularmente, o Netscape define uma API em linguagem C chamada de NPAPI (Netscape Plugin API) [Mozc], que define uma série de funções que o navegador expõe para os *plugins* e que espera que os plugins exponham para serem chamadas pelo navegador. Desta forma, ao se elaborar um plugin compatível com a NPAPI, ele poderá ser utilizado com todos os navegadores compatíveis.

1.3.2 Complementos no Netscape

A arquitetura de complementos (*plug-ins*) utilizada pelos navegadores da família Netscape [Net] é baseada em bibliotecas compartilhadas (*shared libraries*) que o navegador carrega em tempo de execução. As bibliotecas compartilhadas são um conceito bastante antigo no sistema operacional UNIX, já constando por exemplo no System V Release 3 [Arn86]. Como explica Stevens [Ste93, 169], as bibliotecas compartilhadas são uma maneira bastante flexível por meio da qual a aplicação pode carregar rotinas externas em tempo de execução, permitindo portanto o desacoplamento entre o código dessas rotinas e o código da aplicação. Assim, ainda segundo o autor, essas rotinas podem ser inclusive substituídas por outras sem que o aplicativo precise ser compilado ou ligado novamente, desde que seja preservada a mesma interface [Ste93, 170].

Além dos propósitos originais apontados por Stevens [Ste93, 169], entre os quais estão a facilidade de atualização e a possibilidade de manter somente uma cópia das rotinas comuns na memória do sistema operacional, as bibliotecas compartilhadas também introduziram novas possibilidades. Entre elas, está a capacidade de implementar extensões e complementos de navegador por meio da definição de uma interface entre o navegador e esses componentes.

Para que os arquivos gerados por diversos compiladores fossem compatíveis, o UNIX System V definiu um padrão de formato binário [AT97] que impõe uma estrutura pré-estabelecida para o conteúdo dos arquivos executáveis e estabelece uma convenção consistente de chamada de funções. Esta padronização permite que um aplicativo e seus complementos possam ser desenvolvidos separadamente, podendo inclusive ser elaborados com compiladores diferentes, desde que o padrão binário seja seguido. Esse formato ficou conhecido como *ELF - Executable Linkable File*.

Assim, num navegador, cada um desses componentes (*plugins*) é uma biblioteca compartilhada que implementa uma interface pré-estabelecida para a comunicação entre o navegador e o complemento. No caso do Linux, essas bibliotecas compartilhadas são compiladas na forma de arquivos de extensão *.so* (geralmente obtidos por meio da opção *-shared* do compilador GCC [FSF]) e as funções da interface são simplesmente pontos de entrada padrão do formato binário ELF.

1.3.3 Interação entre Complementos e o Navegador

As páginas visualizadas num navegador geralmente vêm expressas no formato HTML, que atualmente se encontra em transição da sua versão 4.0 [RLHJ98] para a versão 5.0 [HH10]. Porém, o conceito de "objetos embutidos" (*embedded objects*) já é consideravelmente antigo e já estava razoavelmente consolidado desde o início da adoção do HTML 4.0, particularmente por meio do elemento `<object>` (e, similarmente, do elemento alternativo `<embed>`), como explica Holzschlag [Hol00, 672,976-978].

Um objeto embutido numa página geralmente apresenta ao usuário um conteúdo dinâmico de um tipo diferente da página como, por exemplo, vídeo, áudio ou documentos em PDF. Há inclusive objetos embutidos que exibem aplicativos completos, como o Adobe Flash [Ado]. Os navegadores modernos, em sua maioria, já trazem a funcionalidade necessária para apresentar uma série de formatos de conteúdo embutido, entre eles os formatos comuns de áudio e vídeo. Para identificar o tipo de conteúdo, os navegadores utilizam o critério do *MIME type*, que era originalmente uma maneira de especificar o tipo de conteúdo em mensagens de correio eletrônico [FB96]. Um *MIME type* consiste numa cadeira de caracteres que especifica, de maneira inambígua, um determinado tipo de conteúdo. Por exemplo, uma página HTML tem *MIME type* `text/html`; uma imagem no formato JPEG tem *MIME type* `image/jpeg` e assim por diante. A entidade responsável por manter a lista oficial dos *MIME types* para os diversos formatos existentes é a Internet Assigned Numbers Authority (IANA) [IN].

O objetivo dos complementos (*plug-ins*) é ampliar o leque de formatos de conteúdo (representados pelos *MIME types*) aceitos pelo navegador. Por isso, uma das informações fundamentais que um complemento precisa fornecer ao navegador é uma relação de quais *MIME types* o complemento é capaz de exibir. Como pode haver vários complementos instalados simultaneamente, os navegadores decidem qual complemento utilizar com base no *MIME type* do conteúdo.

1.4 O Python e o Stackless Python

O Python [Pyt89] é uma linguagem de programação que desfruta de considerável popularidade, com uma base de código madura e mantida por uma comunidade muito ativa. Trata-se de uma linguagem multiparadigma, isto é, adequada a diversos tipos de paradigma de programação, entre eles a orienta-

ção a objeto e a programação funcional.

Como explica Guido van Rossum [Gui03], o criador da linguagem Python, uma das contribuições mais inovadoras da linguagem foi a facilidade com que ela pode ser estendida e embutida, contrariando o design de outras linguagens que adotam o padrão "monolítico." Este fator, aliado à popularidade da linguagem e facilidade de programação, levou-nos a escolher o Python como linguagem para implementação do PAWEB. Afinal, tanto a capacidade de estender quanto a de embutir a linguagem são cruciais: para fornecer o componente cliente é necessário embutir o interpretador num complemento de navegador escrito em C, enquanto que para fornecer a funcionalidade de atores para o código hospedado, é necessário estender a linguagem com novos módulos.

No campo da programação concorrente, embora a linguagem Python forneça suporte para *threads* e primitivas de sincronização por meio de seu módulo `thread` [Pytb], não há suporte nativo para o modelo de atores como existe, por exemplo, em Erlang [Eri]. Porém, como explicamos anteriormente, a infraestrutura necessária para a implementação do modelo de atores está presente.

No entanto, um dos problemas com a implementação de atores por intermédio de *threads* é que há um custo elevado envolvido em sua criação e administração, pois, como explica Tanenbaum [TW97, 56], a manipulação de *threads* exige a criação e manutenção de diversas estruturas no sistema operacional [TW97, 56]. A linguagem Stage [AE09], lida com esse problema alocando um número pequeno de *threads* (tipicamente da ordem de algumas dezenas) e distribuindo atores entre várias *threads* através de algoritmos de balanceamento de carga.

Ao invés de implementar os atores com *threads* ou adotar a implementação do Stage, decidimos optar por uma versão modificada do Python chamada Stackless Python [Chra]. A razão para essa escolha, explicando de maneira sucinta, é que o Stackless Python fornece suporte às chamadas *tasklets*. Como explica Tismer [Tis00], uma *tasklet* é uma espécie de *thread* mais leve gerenciada e escalonada pelo próprio programa ao invés do sistema operacional, e por isso centenas ou milhares delas podem ser criadas sem o impacto proibitivo que seria gerado caso criássemos centenas ou milhares de *threads*. Essa flexibilidade é importante porque num programa típico no modelo de atores, o número de atores pode ser razoavelmente elevado enquanto que a carga de trabalho de cada um será razoavelmente pequena. As *threads* tradicionais estão adaptadas justamente para o cenário inverso: um número reduzido de threads com alta carga de trabalho.

O Stackless Python foi introduzido por Tismer [Tis00] em 2000. Como explica o autor, o Stackless consiste numa alteração do interpretador Python de forma a dissociar a pilha de chamadas do interpretador da pilha de chamadas do programa interpretado. Essa alteração aparentemente pequena permite uma enorme flexibilidade de programação, pois a partir do momento em que o interpretador (em C) não depende de sua própria pilha para manter a pilha do programa executado, passa a ser possível fornecer primitivas para mudar de contexto de pilha, restaurar um contexto anterior, realizar saltos não-locais, entre outros. Como observa o autor, este tipo de flexibilidade permite a implementação de estruturas que dependem de mudanças de contexto, como por exemplo co-rotinas, geradores e *tasklets* [Tis00]. Ao invés de implementar diretamente essas estruturas, Tismer [Tis00] implementa uma única primitiva: a continuação.

Um dos primeiros usos de continuações de que se tem notícia foi no trabalho de van Wijngaarden, em 1966 [vW66]. Porém, o conceito de continuações não se tornou amplamente conhecido imediatamente. Como explica Reynolds [Rey93, 236-237], isto parece ter ocorrido porque o trabalho de Wijngaarden, embora utilize continuações, não as descreve nem enfatiza especificamente. De fato, a partir da década de 1960, os conceitos básicos das continuações foram descobertos independentemente por vários autores e grupos, em grande parte por causa da grande variedade de situações e contextos nos quais são úteis [Rey93].

Se analisarmos as linguagens mais populares da indústria, especialmente na área da programação *web* (PHP, Java, C#, ASP.NET, entre outras), notaremos que a grande maioria delas tem suporte extremamente limitado para a programação funcional. Implementar continuações nessas linguagens é extremamente difícil, visto que em várias dessas linguagens não existe sequer o conceito de função como valor de primeira ordem.

Porém, isto não é uma indicação de que o conceito de continuações seja incompatível ou de pouca utilidade nesse contexto. De fato, Krishnamurthi [Kri07, 148-156] expõe e exemplifica em detalhe a grande afinidade entre o conceito de continuações e o comportamento de aplicações *web* que mantêm um estado.

Explicando sucintamente, uma continuação é uma representação do resto da computação como um procedimento que recebe um argumento [Kri07]. Em outras palavras, é uma forma de gravar o estado atual da computação de um procedimento na forma de um valor que pode ser livremente passado como argumento, devolvido ou atribuído a variáveis. Esse valor permite que a computação "suspendida" seja retomada a qualquer momento, em qualquer

outra parte do programa.

Como explica Tismer [Tis00, 2], a implementação de continuações no Stackless Python exigiu um grande esforço para possibilitar a separação do vínculo entre a pilha do interpretador e a pilha do programa interpretado. Porém, ainda segundo o autor, a existência das continuações como primitiva é suficiente para a implementação dos demais conceitos principais do Stackless.

De fato, o poder conferido ao interpretador pelas continuações permite a execução de uma série de tarefas em paralelo, cada uma das quais com seu contexto de pilha e de chamadas, pois a qualquer momento elas podem ser interrompidas e seu estado pode ser mantido como uma continuação. O escalonamento consiste essencialmente em gerenciar essas continuações, que inclusive podem exceder em muito o número de processadores físicos disponíveis, o que não era possibilitado por implementações anteriores de geradores e co-rotinas em Python [Tis00, 1].

1.5 Trabalhos Relacionados

Por ora, limitamo-nos a mencionar cada um dos trabalhos relacionados e apresentar um breve resumo. Ao final do presente texto, apresentaremos uma comparação mais detalhada entre o PAWEB e esses trabalhos, apontando características comuns e principais diferenças.

Primeiramente, é necessário mencionar novamente a linguagem Erlang, descrita por Armstrong em 2003 [Arm03], que, como explicamos em detalhe anteriormente, é um dos exemplos mais representativos de uma linguagem centrada no modelo de atores. Como explica o autor da linguagem, o Erlang permite, por meio de sua biblioteca OTP, que os atores sejam executados em máquinas diferentes e se comuniquem através de uma rede.

A linguagem Scala [O+03], também mencionada anteriormente, também fornece suporte ao modelo de atores. Porém, como explica Pollak [PV10, 88], os atores não são uma característica nativa da linguagem e sim uma biblioteca. Porém, a flexibilidade da sintaxe da linguagem facilita a integração transparente do uso da biblioteca, flexibilidade essa que é parte da filosofia de "escalabilidade" da linguagem [OAC+04].

Também existe o arcabouço Lift para programação *web* em linguagem Scala, que, como explica Ghosh [GV09], foi inspirado no fato de que paradigmas funcionais se adaptam melhor à *web* do que a programação imperativa tradicional. É possível a realizar programação orientada a atores em Lift, como

demonstra o aplicativo de exemplo apresentado por Pollak et al. [PV10].

A linguagem SALSA, introduzida por Varela e Agha [VA01] em 2001, é uma linguagem orientada a atores de propósito geral que, segundo os autores, foi especialmente projetada para facilitar o desenvolvimento de aplicações distribuídas dinamicamente reconfiguráveis. Numa aplicação em SALSA, atores podem migrar entre máquinas durante sua execução.

Ayres e Eisenbach [AE09] introduziram a plataforma Stage em 2009. Como explicam os autores, o Stage é uma modificação do Python que introduz algumas novas construções sintáticas para dar suporte às abstrações do modelo de atores, procurando também aproximar esse modelo da metodologia da orientação a objetos. O Stage também é capaz de distribuir atores numa rede de forma a realizar o balanceamento de carga, assim como o SALSA.

Podemos mencionar também o Candygram, proposto por Hobbs [Hob04]. Trata-se de um módulo para a linguagem Python que implementa a concorrência nos moldes da linguagem Erlang [Eri]. Como explica o autor, o pacote permite que as *threads* Python enviem e recebam mensagens com uma semântica quase idêntica à do Erlang, permitindo portanto a implementação do modelo de atores de uma maneira bastante intuitiva para programadores já habituados àquela linguagem.

Capítulo 2

Arquitetura e Funcionamento

O foco deste capítulo é o funcionamento do sistema do ponto de vista do usuário, isto é, do desenvolvedor que escreve o código que será executado na plataforma. Explicaremos a arquitetura e a maneira como desenvolvedores podem interagir com os módulos e funções que compõem a interface de programação de aplicações (*API*) do sistema. No capítulo seguinte, descreveremos a implementação do sistema em si e forneceremos uma explicação detalhada sobre as técnicas utilizadas.

2.1 Visão Geral da Arquitetura

A plataforma PAWEB consiste numa série de componentes, alguns dos quais residentes do lado do cliente e outros do lado do servidor. Primeiramente, uma vez que o PAWEB é uma plataforma de execução de aplicativos, chamamos de "aplicativo hóspede" ou "programa hóspede" o código fornecido pelo programador que utiliza a plataforma. Esse código tipicamente consistirá num programa em Python que cria um ou mais atores, alguns dos quais no servidor e outros nos clientes, e que permanece em execução por tempo indeterminado. Mesmo quando não há clientes conectados, os atores residentes no servidor normalmente permanecem ativos aguardando mensagens ou até mesmo efetuando tarefas administrativas em segundo plano.

Conforme usuários acessam o sistema, surgem atores do lado dos clientes para trocar mensagens com os atores no servidor. Esses atores existentes nos clientes terminam quando os clientes se desconectam. O servidor da plataforma PAWEB, que chamamos PAWEB Server, consiste num programa escrito em Python que funciona como contêiner de aplicações, isto é, contém e gerencia um ou mais aplicativos-hóspedes, que por sua vez também são programas em

Python.

O servidor também fornece um módulo de código (biblioteca) chamado PAWEB Server Runtime (abreviado PAWEB-SRT), que implementa a funcionalidade necessária para utilizar os recursos da plataforma. Entre esses, os mais importantes são a criação e o gerenciamento dos atores e a infraestrutura para a transmissão de mensagens entre atores qualquer que seja sua localização relativa (mensagens internas dentro do servidor, dentro do cliente ou entre o servidor e o cliente). O PAWEB-SRT encapsula todos os detalhes da conexão e do protocolo de transmissão de dados, fornecendo ao aplicativo hóspede uma interface transparente por meio da qual é possível interagir com quaisquer atores de maneira simples e direta, sem a necessidade de lidar com detalhes como conexões, *sockets*, protocolo HTTP, representação binária de dados, erros de conexão, entre outros.

Como explicamos anteriormente, os navegadores da família Netscape [Net], incluindo o Mozilla Firefox [Mozb], permitem a utilização de complementos (*plugins*). Esses complementos são instalados como bibliotecas compartilhadas, isto é, módulos de código que se ligam dinamicamente à aplicação fornecendo novos dados e rotinas, permitindo estender ou alterar a aplicação [Ste93, 170]. No lado do cliente, o PAWEB fornece sua funcionalidade na forma de um complemento de navegador que instanciará a máquina virtual Python na qual os atores do aplicativo hóspede serão executados. Portanto, essa máquina virtual também fornecerá uma biblioteca similar à PAWEB-SRT, e que na verdade compartilhará com a mesma uma boa parte de seu código fonte. Chamamos essa biblioteca de PAWEB Client Runtime (e abreviamos PAWEB-CRT). Complementos para outros navegadores (fora da família Netscape) poderão ser desenvolvidos em projetos futuros, como sugerimos no final deste texto.

Existe ainda um último componente que terá de ser implementado, que é o módulo em C que fará a comunicação entre a PAWEB-CRT e o navegador em si. Chamamos esse componente de PAWEB System Interface (e abreviamos PAWEB-SIF), e é por meio dele que o código cliente faz as chamadas necessárias para receber a entrada do usuário, desenhar elementos gráficos na tela do plugin e requisitar outros serviços que o navegador disponibiliza. A Figura 2.1 ilustra de maneira resumida a arquitetura da plataforma, utilizando as abreviações aqui convencionadas.

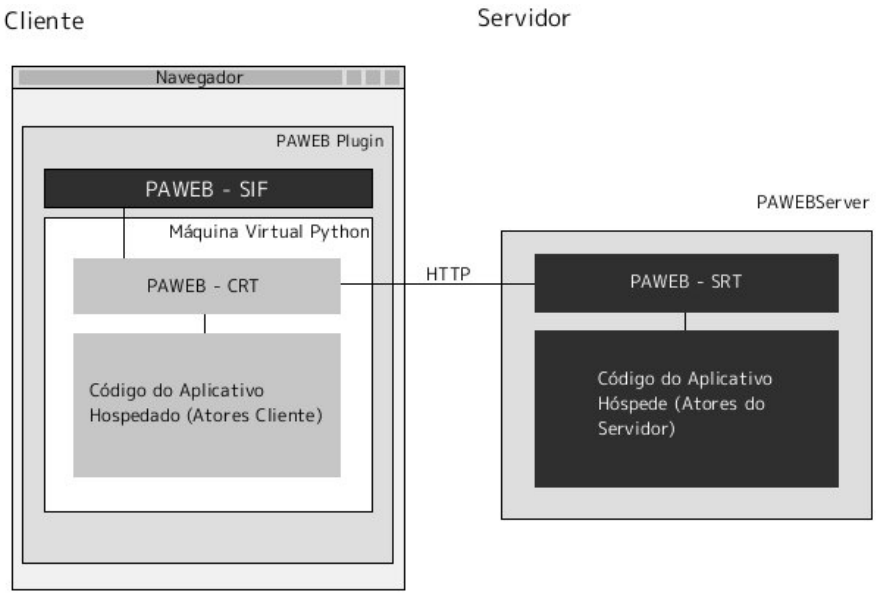

Figura 2.1: *Arquitetura da Plataforma*

2.2 Interface com o Navegador

Na *web*, os tipos de conteúdo são identificados por códigos chamados *MIME types*, definidos pela RFC 2045 [FB96]. Para ser considerada sintaticamente correta, uma resposta com corpo no protocolo HTTP deve sempre incluir um cabeçalho com a declaração do *MIME type* [FGM+99]. O consumidor dos dados (por exemplo, um navegador) utiliza este código para determinar a forma como deve processá-los ou exibí-los. Portanto, para implementar o complemento, primeiramente precisamos definir qual será o *MIME type* que representará um aplicativo PAWEB. O prefixo `application/x-` indica que um determinado *MIME type* representa um formato que não é oficialmente reconhecido pela IANA [IN], a autoridade oficial que, entre outras atribuições, funciona como registro centralizado para os *MIME types* considerados oficiais na indústria [FB96].

Uma vez que o PAWEB é um projeto experimental e naturalmente não desfruta de um registro na IANA, devemos utilizar esse prefixo. O restante do código é livre, então decidimos pela seguinte forma:

```
application/x-paweb-client-code
```

Para informar o navegador que este é o *MIME type* a que o complemento dá suporte, implementamos a função `NP_GetMIMEDescrption` da NPAPI [Moza] de forma a devolver este valor. A partir desse momento, quando o navegador se

deparar com uma página nesse formato ou que contenha um objeto embutido nesse formato, o nosso complemento será instanciado e executado.

Os complementos podem ser utilizados para implementar componentes específicos dentro de páginas HTML ou para implementar a página inteira. O primeiro caso ocorre quando uma página HTML tradicional (formato `text/html`) contém uma tag `<object>` ou `<embed>`. Nesse caso, a própria *tag* indica o *MIME type* do conteúdo embutido. Já no caso da página inteira, o servidor web utiliza o campo `Content-Type` da resposta HTTP para especificar o *MIME type* apropriado, e nesse caso não há nenhum "envelope HTML."

Uma vez que as funções da NPAPI [Moza] são uma interface de baixo nível apropriada somente para a linguagem C, implementamos funções em C e um componente em Python que visam encapsular a funcionalidade da NPAPI de uma forma compatível com a arquitetura da plataforma. Essa camada de compatibilidade, ilustrada na Figura 2.1, recebe o nome de PAWEB-SIF.

2.3 Atores no PAWEB

Como já explicamos anteriormente, o trabalho original de Hewitt, Bishop e Steiger [HBS73] estabelece o modelo de atores como um formalismo teórico independente de implementação concreta, o que resulta numa grande liberdade de implementação em diversas plataformas e linguagens. Como explica Agha [Agh86, 12], o cerne do modelo consiste na existência dos atores como unidades de execução independentes cujo comportamento é regido pelo recebimento de mensagens. Cada ator pode, em resposta a uma mensagem recebida, enviar mensagens a outros atores, criar novos atores, e mudar seu comportamento para reagir de forma diferente a novas mensagens. Essa última característica, vital para diferenciar o modelo de atores da simples aplicação de funções matemáticas, é conhecida como "sensibilidade ao histórico" ([Agh86, 13]).

Uma implementação minimalista poderia, portanto, fornecer somente essas funcionalidades. Embora essa implementação seja em teoria mais simples, a conveniência de utilização do ponto de vista do desenvolvedor poderia ser bastante prejudicada porque faltariam uma série de estruturas e construções provenientes de linguagens existentes. Certamente essas construções faltantes poderiam ser implentadas a partir das primitivas fornecidas, mas tomamos a decisão de torná-las parte da plataforma para maior facilidade. Por isso, além de implementar o modelo teórico como descrito por Agha, optamos por

implementar algumas características convenientes provenientes da linguagem Erlang [Eri], adaptando-as à linguagem Python e ao ambiente *web*. A seguir explicaremos algumas das características da representação de atores que escolhemos implementar no PAWEB.

2.3.1 Representação

No PAWEB, um ator é representado por um objeto Python. Esse objeto é opaco ao usuário, isto é, só deve ser utilizado de acordo com a interface estabelecida pela plataforma e o usuário não deve manipular o conteúdo desse objeto.

2.3.2 Comportamento

O comportamento de um ator é dado por uma função Python. Portanto, sempre que o comportamento de um ator tem que ser especificado, isto é feito por meio de uma referência a uma função. Por exemplo, para criar um novo ator, deve-se especificar a função que esse ator executará quando for iniciado.

2.3.3 Criação de Atores

Um ator pode criar outros atores e especificar seu comportamento inicial. De fato, com a exceção dos atores iniciais, todo outro ator no sistema será necessariamente criado por um outro ator. A criação de atores é assíncrona, isto é, não há garantia de que o ator será executado imediatamente no momento em que for criado; caso seja necessário fazer qualquer tipo de sincronização com os atores recém-criados, deve-se utilizar o envio e recepção de mensagens. O comportamento inicial do novo ator é especificado pelo ator que o criou, mas esse comportamento não é perpétuo, pois uma das características fundamentais de atores é a possibilidade de mudar de comportamento durante a execução.

2.3.4 Mensagens

Ainda que o modelo teórico não especifique a natureza exata de uma mensagem, é desejável que as mensagens não estejam limitadas a tipos primitivos, mas possam conter estruturas complexas e hierárquicas de dados. Por isso, no PAWEB, qualquer objeto pode ser usado como mensagem desde que seja "seriável", isto é, possa ser representado como uma cadeia de *bytes*. Isto ocorre naturalmente com os tipos nativos do Python e com suas composições na

forma de estruturas de dados embutidas na linguagem. Porém, como explicaremos em breve na implementação, uma limitação significativa é que funções dinâmicas não são "seriáveis" e portanto não podem ser utilizadas como parte de mensagens.

2.3.5 Envio de Mensagens

Um ator pode enviar mensagens assincronamente a outro ator cujo endereço possua. O envio pode ocorrer a qualquer momento durante o ciclo de vida do ator exceto, naturalmente, quando o ator está bloqueado aguardando o recebimento de uma mensagem.

2.3.6 Recebimento de Mensagens

Assim como ocorre no Erlang, no PAWEB a operação de recebimento de mensagens pode causar o bloqueio do ator até que uma mensagem seja recebida. Também com base no Erlang, o recebimento de mensagens é seletivo e feito por meio do casamento de padrões, que explicaremos posteriormente.

Quando um ator deseja receber mensagens, efetua uma chamada à API da plataforma especificando os padrões de mensagens que espera e qual o novo comportamento que o ator deverá assumir em resposta a cada tipo de mensagem recebida. Esse comportamento, como explicamos anteriormente, sempre é especificado como uma referência a uma função Python.

2.4 Pressupostos

Os atores no PAWEB operam sob alguns pressupostos. Primeiramente, cada ator possui um endereço globalmente único, que o identifica em meio a todos os atores existentes, ainda que distribuídos por diversas máquinas. Esta suposição é necessária para que não haja ambiguidade na entrega de mensagens e para que seja possível implementar a transparência de localização, que explicaremos em maior detalhe a seguir.

Outro pressuposto é que o envio de mensagens é assíncrono, isto é, a execução de um ator não será interrompida ao enviar uma mensagem, mesmo que o ator destinatário não a receba imediatamente. Uma consequência desse comportamento é a necessidade de que cada ator tenha uma "caixa de correio" associada, na qual são armazenadas as mensagens que lhe foram enviadas e ainda não foram processadas.

No PAWEB, também garantimos a *transparência de localização*, que é o princípio de que os atores se comunicam entre si de maneira transparente independentemente da máquina física em que residem. Ao se comunicar com outro ator por meio do envio ou recepção de mensagens, o fato de que o outro ator reside numa máquina remota e não na máquina local não altera a forma como o desenvolvedor deve escrever o código: a plataforma se encarrega de fornecer a infraestrutura necessária para que a mensagem seja transmitida à máquina correta.

Embora a plataforma não forneça nenhuma garantia sobre a natureza exata de um ator, é garantido que a implementação de um ator é um processo leve, possibilitando a criação de milhares de atores sem que isso cause um impacto significativo no desempenho do sistema. Isso é importante porque em sistemas onde atores são implementados por meio de estruturas mais pesadas como processos ou *threads*, os programas não podem tomar a liberdade de criá-los em grande quantidade.

2.5 Criação de Atores

No PAWEB, a criação de um ator é feita com uma chamada ao sistema, que recebe uma referência ao código que o ator deverá executar. A chamada devolve uma referência ao ator criado, de maneira muito semelhante à primitiva `spawn` da linguagem Erlang [Arm07, 142]:

```
1 import paweb
2 def my_fun(me):
3   # codigo aqui
4   print "Ola!"
5
6 my_actor = paweb.actor(my_fun)
7 # execucao do ator original continua aqui
```

O ato de criar um ator não afeta a execução do ator original. O ator original também não pode fazer suposições sobre o momento em que o código do ator recém-criado (nesse caso, a função `my_fun`) será executado. Convém lembrar que a função inicialmente especificada como comportamento do ator não é necessariamente permanente: o ator poderá mudar de comportamento durante sua execução. Também convém lembrar que, como explicaremos em breve, a referência ao ator não é o mesmo que o endereço do ator: para obter o endereço de um ator a partir da referência, usa-se o campo `gid`:

```
1 print "O endereco do ator e " + my_actor.gid
```

2.6 Chamadas de Cauda (Tail Calls)

Uma chamada de cauda (tail call) ocorre quando a última instrução executada por uma função é uma chamada a outra função. Como explica Armstrong [Arm07, 157], certos compiladores convertem esse tipo de chamada numa simples instrução de salto (*jump*) para o código da função chamada, uma vez que não é necessário preservar o contexto de execução para um eventual retorno à função original. Assim, esse tipo de chamada não reserva o espaço de pilha que uma chamada de função normalmente utilizaria, permitindo inclusive a implementação de chamadas recursivas que podem ocorrer um número arbitrariamente grande de vezes sem que haja acúmulo de contextos na pilha e potencial exaustão do espaço, fenômeno conhecido como "estouro de pilha." A linguagem Erlang garante que chamadas desse tipo serão implementadas deste modo, de maneira que o programador pode tomar essa otimização como pressuposto e portanto pode implementar recursões teoricamente infinitas sem se preocupar com o estouro de pilha.

Porém, não existe tal garantia em Python. De fato, na implementação atual (versão 2.6), o compilador não efetua essa otimização, de maneira que foi necessário implementar um mecanismo alternativo.

Assim, para efetuar uma chamada de cauda, o desenvolvedor do aplicativo hospedado não deve chamá-la diretamente. Ao invés disso, deve utilizar o método `become()`:

```
1 import paweb
2
3 def funcao_a(me):
4   # faz algo
5   ...
6
7   # chamada de cauda a funcao_b
8   me.become(funcao_b)
9
10 def funcao_b(me):
11   # faz algo
12   ...
```

A chamada a `become()` garante que não será gerado resíduo de pilha na chamada. Essa chamada pode ser utilizada para chamar outra função ou

mesmo a função original recursivamente.

Como esperado, qualquer código que suceda à chamada `become()` nunca será executado, uma vez que o contexto atual será descartado:

```
1 import paweb
2
3 def comportamento_a(me):
4     ...
5     me.become(bar)
6     print "ola..."  # INCORRETO: nunca sera executado
```

2.7 Endereço de Atores

Como explicamos anteriormente, um dos pressupostos do sistema é que cada ator possui um endereço globalmente único. Na linguagem Erlang, não há distinção sintática entre o endereço de um ator e a referência ao ator, isto é, para comunicar o endereço de um ator numa mensagem, basta utilizar a referência ao próprio ator, como ilustra o código de exemplo abaixo.

```
1 f() -> AtorA ! { ola, AtorB };
```

A função `f` envia uma mensagem ao ator `AtorA` que consiste numa tupla formada pelo átomo `ola` e pelo ator `AtorB`. Porém, como esperado, não é o ator em si que é embutido na mensagem, mas sim uma referência a ele, de maneira que o destinatário possa usar essa referência para se comunicar com o ator. Caso o ator fosse passado por valor, isso resultaria na criação de um novo ator diferente do original, que não é o propósito do código. Em outras palavras, o conteúdo da mensagem não é o ator em si, mas uma maneira de referenciá-lo unicamente, e a isso damos o nome de "endereço".

Porém, no PAWEB, tivemos que adaptar essa sintaxe porque as mensagens são passadas inteiramente por valor, já que dependem de "seriação" para possível transmissão a outras máquinas na rede. Assim, caso usássemos uma sintaxe semelhante, resultaria que o ator referenciado seria seriado como parte da mensagem, que não é o nosso propósito.

Assim, tivemos que introduzir uma distinção sintática entre o objeto e o endereço. Quando desejamos o endereço de um ator ao invés do ator em si, utilizamos o campo `gid` do objeto, como exemplificado abaixo.

```
1 import paweb
2
3 def f():
```

```
4    paweb.send(ator_a, ("ola", ator_b.gid))
```

A diferença é que ao invés de especificar simplesmente `ator_b`, utilizamos `ator_b.gid` para especificar que desejamos embutir na mensagem o endereço (também chamado *globally unique identifier, GID*) do ator.

O tipo do endereço é uma *string* do Python, isto é, uma cadeia de caracteres que identifica unicamente o ator. Assim, endereços podem ser livremente embutidos em mensagens ou até armazenados em arquivos ou outros meios.

2.8 Envio de Mensagens

Para enviar uma mensagem, utiliza-se a função `paweb.send` da API.

```
1 import paweb
2 ...
3 paweb.send(dest, mensagem)
```

Neste exemplo, presume-se que as variáveis `dest` e `mensagem` foram inicializadas anteriormente. A variável `dest` seria uma referência ao ator destinatário (ou seu endereço) e `mensagem` seria a mensagem a enviar, que poderia ser qualquer objeto Python ou estrutura de dados seriável. O exemplo a seguir envia uma mensagem que contém um nome, um sobrenome e uma lista de autores favoritos:

```
1 import paweb
2
3 paweb.send(dest, (nome, sobrenome,
4       ["George Orwell", "Isaac Asimov", "Gabriel Garcia Marquez
          "]))
```

Observe que, para maior conveniência, `paweb.send` aceita tanto uma referência ao ator quanto o seu endereço como seu primeiro parâmetro. Ou seja, é indiferente utilizar `dest` ou `dest.gid`.

O envio de mensagens é assíncrono. Assim como ocorre no Erlang, a execução do ator atual não é bloqueada mesmo se o ator destinatário não receber imediatamente a mensagem, o que é diferente do que ocorre com *tasklets* e *channels* no Stackless Python ([Tis00]).

Normalmente, a entrega de mensagens não garante nenhuma ordenação específica, mas no PAWEB também implementamos uma garantia parcial da ordem das mensagens enviadas entre atores. Essa garantia, embora não estritamente necessária, torna muito mais conveniente a programação. A garantia

pode ser expressa da seguinte forma: se um ator A envia uma mensagem M_1 a um ator B e posteriormente A envia a mensagem M_2 ao mesmo ator B, então garante-se que quando o ator B efetuar o recebimento de mensagens especificando o padrão P para o recebimento, e se M_1 e M_2 ambas casam com o padrão P, então, a mensagem M_1 será recebida antes da mensagem M_2.

Em outras palavras, excluindo o efeito de filtragem ocasionado pelo uso do casamento de padrões, as mensagens enviadas pelo mesmo remetente ao mesmo destinatário serão recebidas na ordem em que foram enviadas. Por outro lado, nenhuma garantia é oferecida com relação à ordem da entrega de mensagens entre pares de atores diferentes.

O envio de mensagens remotas é transparente para o desenvolvedor, ou seja, a sintaxe de envio é a mesma independente do destinatário ser local ou remoto. Por meio do endereço, a infraestrutura do PAWEB determinará como realizar o envio. Forneceremos mais detalhes sobre o funcionamento desse mecanismo no Capítulo 3.

2.9 Recebimento de Mensagens

Para receber mensagens de outros atores, utiliza-se o método `receive()`, que foi inspirado na primitiva `receive` do Erlang. Essa primitiva permite que vários padrões de mensagens e comportamentos correspondentes sejam especificados; quando o ator receber uma mensagem que casa com um dos padrões indicados, o comportamento correspondente àquele padrão será executado ([Arm07]). A sintaxe típica dessa primitiva na linguagem Erlang é a seguinte:

```
1 receive
2    Pattern1 -> Expressions1;
3    Pattern2 -> Expressions2;
4    ...
5 end
```

No momento em que a primitiva `receive` é executada, a execução do ator fica bloqueada até que seja recebida uma mensagem que casa com um dos padrões indicados (`Pattern1`, `Pattern2`, etc). Quando isso ocorre, as expressões correspondentes (`Expressions1`, `Expressions2`) são executadas. Seguimos a mesma filosofia na implementação do método `receive()` no PAWEB, porém adaptando-a à sintaxe do Python.

```
1 def f(me):
```

```
2    me.receive(padrao_1, fun_1, [],
3                padrao_2, fun_2, [],
4                padrao_3, fun_3, [],
5                ...
6                )
7
8  def fun_1(me):
9    ...
10
11 def fun_2(me):
12   ...
```

Os padrões são expressados como objetos Python e os comportamentos correspondentes são referências a funções que serão executadas quando cada padrão de mensagem for recebida. O parâmetro `me` é uma referência ao próprio ator (similar à função primitiva `self()` do Erlang).

No PAWEB, um *padrão* é definido da seguinte maneira:

1. um valor escalar fixo "seriável" é um padrão;
2. uma variável livre (expressada por um tipo de objeto especial no PAWEB, como explicaremos posteriormente) é um padrão;
3. uma lista (`list` em Python) de padrões é um padrão;
4. uma tupla (`tuple` em Python) de padrões é um padrão;
5. um dicionário (`dict` em Python) que mapeia *strings* para padrões é um padrão

Isto torna possível a existência de padrões hierárquicos complexos. Definimos o casamento de um padrão P com um valor V da seguinte maneira:

1. se P é um escalar fixo, então V casa com P se e só se $V = P$, pela definição de igualdade entre tipos escalares em Python;
2. se P é uma variável livre, P casa com V (para qualquer valor de V), e V será atribuído (vinculado) à variável como resultado da operação de casamento de padrões, como descreveremos a seguir;
3. se $P = [P_1, \ldots P_n]$ é uma lista, então V casa com P se e só se V também é uma lista de n elementos $V = [V_1, V_2, V_3, \ldots V_n]$ onde P_i casa com V_i para cada $i = 1 \ldots n$;

4. se $P = (P_1, \ldots P_n)$ é uma tupla, então V casa com P se e só se V também é uma tupla de n elementos $V = (V_1, V_2, V_3, \ldots V_n)$ onde P_i casa com V_i para cada $i = 1 \ldots n$;

5. se P é um dicionário que mapeia a chave k_1 para o padrão P_1, k_2 para o padrão P_2 e assim por diante até P_n, então dizemos que V casa com P se V também é um dicionário com o mesmo conjunto $\{k_1, \ldots k_n\}$ de chaves e, considerando que V mapeia cada k_i para um valor v_i, se v_i casa com P_i para cada $i = 1, \ldots n$.

Assim como ocorre no Erlang, o PAWEB admite a existência de "variáveis livres" dentro dos padrões. Essas variáveis livres casam com qualquer valor e servem para capturar as partes do valor que lhe correspondem para processamento posterior. No Erlang, essas variáveis são chamadas *não-vinculadas* [Arm07, 29], pois são aquelas que não têm definição anterior no escopo e aparecem por primeira vez dentro do padrão, em oposição às *vinculadas*, que já têm um valor pré-existente.

No PAWEB, uma variável livre num padrão é marcada pela sintaxe especial `paweb.bindvar` como explicaremos a seguir.

O seguinte trecho de código demonstra o recebimento de mensagens baseado em casamento de padrões num servidor de "bate-papo":

```
1 def recepcionista(me):
2     print("Esperando mensagens...")
3     me.receive( \
4         ("OI", paweb.bindvar("ator"), paweb.bindvar("nome")),
              cadastra, [],
5         ("MSG", paweb.bindvar("quem"), paweb.bindvar("msg")),
              fala, [],
6         paweb.bindvar("mensagem"),
              erro, [])
```

Este trecho de código especifica três padrões para efetuar o recebimento de mensagens. O primeiro padrão consiste numa tupla de três elementos, o primeiro dos quais é a cadeia de caracteres literal "OI", que casará somente se a tupla recebida tiver essa mesma cadeia de caracteres na sua primeira posição; o elemento seguinte é uma variável livre chamada `ator` e em seguida temos outra variável livre chamada `nome`. Essas variáveis receberão os dados existentes na parte correspondente do valor com o qual o padrão casar. No PAWEB, as variáveis livres são marcadas pela chamada `paweb.bindvar`, já que não existe sintaxe nativa no Python para tal.

O segundo padrão é semelhante, mas a cadeia de caracteres literal é "MSG". Esse uso de cadeias de caractere fixas para marcar o tipo da mensagem é semelhante ao uso de *átomos* no Erlang, que são valores usados para representar constantes não numéricas de forma semelhante aos `defines` ou `enums` do C ([Arm07, 33]).

O terceiro padrão consiste somente numa variável livre. Portanto, esse terceiro padrão casará com qualquer mensagem recebida que não case com nenhum dos dois padrões que o antecedem. Implementar um padrão desse tipo como mecanismo de segurança para mensagens espúrias é uma boa prática, pois permite detectar erros de programação que causam o envio de mensagens que não estão devidamente formatadas.

Cada padrão mencionado na chamada ao método `receive` vem acompanhada de uma referência a uma função que determinará o comportamento do ator caso seja recebida uma mensagem que case com o padrão. Essas funções recebem sempre uma referência ao próprio ator como primeiro parâmetro e também recebem os valores das variáveis livres em seus outros parâmetros, como exemplifica o código abaixo.

```
1 def cadastra(me, ator, nome):
2   # cadastra o cliente.
3   # ...
4
5 def fala(me, quem, msg):
6   # o cliente enviou uma fala
7   # ...
8
9 def erro(me, mensagem):
10  # alerta sobre mensagem nao reconhecida
11  # ...
```

Caso nenhuma mensagem recebida case imediatamente com algum dos padrões passados ao método `receive()`, o ator permanecerá bloqueado aguardando mensagens até que receba uma mensagem que case com algum dos padrões. Esse é o mesmo comportamento implementado pelo Erlang [Arm07, 154]. A chamada ao método `receive` nunca devolve o controle ao chamador, porque fará uma "chamada de cauda" a uma das funções, sem deixar qualquer resíduo na pilha.

2.10 Inicialização

A inicialização do sistema ocorre como qualquer programa em Python:

```
1 import paweb
2
3 def main():
4   # codigo aqui
5   ...
6
7 if __name__ == "__main__":
8   main()
```

A função `main()` é responsável por inicializar o PAWEB, criar os primeiros atores do sistema e efetuar, ao final, a chamada `paweb.run()`, que manterá sistema em execução até que todos os atores terminem. A estrutura da função `main()` difere um pouco dependendo se o código é o do servidor ou o do cliente.

2.10.1 Inicialização do Cliente

No caso do cliente, uma estrutura típica de inicialização seria a seguinte:

```
1 import paweb
2
3 def main():
4   # inicializacao
5   paweb.init("client", "192.168.0.2", 8765)
6
7   # cria atores iniciais
8   ator1 = paweb.actor(fun_1, [])
9   ator2 = paweb.actor(fun_2, [])
10  ...
11
12  # permanece em execucao ate o termino de todos os atores
13  paweb.run()
14
15 if __name__ == "__main__":
16  main()
```

A chamada `paweb.init()` efetuará a conexão com o servidor, cujo endereço e porta, nesse exemplo, são respectivamente `192.168.0.2` e `8765`. Embora estejamos especificando o endereço e porta do servidor, a conexão não ocorre imediatamente, mas somente quando há de fato uma mensagem a enviar ao servidor.

Após a inicialização, o cliente tipicamente criará um ou mais atores por meio de chamadas a `paweb.actor()` e então deverá chamar `paweb.run()` para que o sistema entre em execução contínua até o término de todos os atores.

2.10.2 Comunicação com o Servidor

Como o PAWEB implementa a transparência de localização, o cliente pode se comunicar com o servidor sem nenhuma providência especial, bastando enviar uma mensagem a um ator que existe do lado do servidor exatamente como faria para enviar uma mensagem a um ator local. Porém, como explicamos anteriormente, implementamos no PAWEB o conceito de *localidade*, que significa que um ator só pode se comunicar com os atores cujos endereços conheça [Agh86, 150]. Inicialmente, um cliente PAWEB conhece apenas o endereço de um ator no servidor: o do ator recepcionista. Esse endereço é fixo e pode ser expresso pela constante `paweb.RECEPTIONIST`.

O conceito de um "recepcionista" é um dos mecanismos mais poderosos do modelo de atores. Uma das técnicas que esse mecanismo possibilita é a implenetação do encapsulamento ou a composição de funcionalidades de atores, possibilitando a criação de sistemas complexos inteiramente modulares [Agh86, 150]. Nossa escolha pela aplicação desse conceito no PAWEB deve-se ao fato de que é natural que o servidor de uma aplicação web seja um componente encapsulado com uma interface bem definida com o componente cliente, e portanto a escolha de fixar um ator recepcionista como ponto inicial de comunicação entre o servidor e o cliente é natural.

Isto não significa, porém, que o ator recepcionista deva ser necessariamente o único ponto de comunicação entre o cliente e o servidor. Nossa plataforma simplesmente estabelece que o recepcionista é o canal inicial. Já que qualquer ator pode enviar mensagens a qualquer outro ator cujo endereço conheça, não há nenhum impedimento técnico ao cliente obter outros endereços de atores que residam do lado do servidor para se comunicar diretamente com eles sem a necessidade do ator recepcionista.

Se o cliente deseja enviar uma mensagem ao ator recepcionista do servidor, basta utilizar o método `send`:

```
1 paweb.send(paweb.RECEPTIONIST, mensagem)
```

Convém lembrar que nenhum ator do servidor será automaticamente notificado do fato de que um novo cliente se conectou ao servidor. Ao invés disso,

é responsabilidade do cliente enviar uma mensagem ao ator recepcionista para notificar o servidor da sua existência.

2.10.3 Inicialização do Servidor

A inicialização do servidor é similar à do cliente. Porém, ao invés de efetuar uma conexão, o servidor abre um *socket* para aguardar conexões de clientes. O servidor deve também criar e registrar o ator recepcionista, que terá um endereço fixo e será o único endereço do servidor conhecido pelos clientes recém-conectados. O código do servidor tipicamente apresentará a seguinte estrutura:

```
1 import paweb
2
3 def main():
4   # inicializacao: servidor
5   paweb.init("server", "", 8765)
6
7   # cria o ator recepcionista
8   ator_recep = paweb.actor(recepcionista, [])
9
10  # cria outros atores, se necessario
11  ator1 = paweb.actor(fun_1, [])
12  ator2 = paweb.actor(fun_2, [])
13
14  # inicia o servico, indicando o ator recepcionista
15  paweb.start_service(ator_recep)
16
17  # permanece em execucao ate o termino de todos os atores
18  paweb.run()
```

A chamada a `paweb.init` inicializa o servidor e especifica qual o endereço e porta no qual ele receberá as conexões dos clientes. Em seguida, criamos o ator que será o ator recepcionista (`ator_recep`). O servidor tipicamente criará outros atores também conforme necessário (`ator_1`, `ator_2`, etc).

Quando o servidor está preparado para receber mensagens de clientes, deve efetuar a chamada a `paweb.start_service`, especificando qual será o ator recepcionista. A partir desse ponto, basta chamar `paweb.run()` para que o sistema entre em execução contínua até que todos os atores terminem.

2.11 Programa de Exemplo: Bate-Papo

A título de exemplo para melhor ilustrar a API do PAWEB, apresentamos agora um programa simples de bate-papo. Esse programa contém dois componentes: servidor e cliente. O servidor é responsável por aguardar mensagens de clientes e manter um registro de clientes a ele conectados. Sempre que um cliente envia uma mensagem, o servidor replicará a mensagem e entregá-la-á a todos os clientes. O papel do cliente é conectar-se ao servidor, exibir as mensagens recebidas e também enviar as mensagens digitadas pelo usuário.

2.11.1 Código do Cliente

Reproduzimos abaixo o código do cliente, que comentaremos em maior detalhe a seguir.

```
1 import paweb
2 import sys
3 from paweb.log import say
4
5 nome = "?"
6
7 def main():
8     global nome
9     print("Qual e o seu nome?")
10    nome = sys.stdin.readline()[:-1]
11    print("Oi, %s. Conectando voce ao servidor..." % (nome))
12    paweb.log.init("client.log",True)
13    paweb.init("client", "localhost", 8765)
14    print("Voce esta conectado.")
15    print("ID da sua instancia: %s" % (paweb.get_node_id()))
16
17    ator_escutador = paweb.actor(escutador, [])
18    ator_falante   = paweb.actor(falante, [])
19
20    paweb.send(paweb.RECEPTIONIST, ("OI", ator_escutador.gid,
          nome))
21    paweb.run()
22
23 def escutador(me):
24    me.receive( \
25       ("MSG", paweb.bindvar("falante"), \
26                paweb.bindvar("mensagem")),  \
27                imprime_fala, [], \
28        paweb.bindvar("coisa"), erro, [])
```

```
29
30 def imprime_fala(me, falante, mensagem):
31     print("\u001b[1;32m<%s>\u001b[0;32m %s\u001b[0m" % (falante,
           mensagem))
32     me.become(escutador)
33
34 def erro(me, coisa):
35     print("BUG: mensagem inesperada: %s" % repr(coisa))
36     me.become(escutador)
37
38 def falante(me):
39     fala = paweb.readline()
40     paweb.send(paweb.RECEPTIONIST, ("MSG",nome,fala))
41     me.become(falante)
42
43 main()
```

O código inicia perguntando ao usuário o seu nome, que será usado na comunicação (linha 10). Em seguida, inicializa o PAWEB e efetua a conexão com o servidor (linha 13). Após feita a inicialização, o código cria dois atores (linhas 17 e 18): o ator `ator_escutador`, que terá a função de receber mensagens enviadas pelo servidor e o `ator_falante`, que terá a função de enviar as mensagens digitadas pelo usuário ao servidor. Em seguida, o código de inicialização envia uma mensagem de *handshaking* ao ator de recepção do servidor (linha 20) contendo o endereço do ator "escutador" e o nome do usuário. A chamada a `paweb.run()` coloca o sistema em execução.

Observando o código do ator "falante" (linhas 38-41), observe que o comportamento desse ator é bem simples: lê uma linha do teclado, envia a linha para o servidor (identificando-a com o nome do usuário) e executa a mesma função novamente. Note o uso de `me.become()`, método que efetua uma chamada de cauda sem deixar resíduo de pilha. Este uso seria equivalente a usar um laço `while`, mas a sintaxe é mais clara desta forma, e mais similar a construções típicas no Erlang.

Um aspecto importante a relembrar nesse contexto é que a mensagem enviada na linha 40 é uma mensagem a um ator remoto. Porém, devido à transparência de localização do PAWEB, não há necessidade de nenhuma mudança na sintaxe da chamada.

O papel do ator "escutador", cujo comportamento é dado pela função `escutador()` nas linhas 23 a 27, é receber mensagens do servidor e imprimí-las na tela. A cada vez ele que recebe uma mensagem com o padrão apropriado, a fun-

ção `imprime_fala` é executada. Observe como estamos realizando aqui uma chamada de cauda de volta à função `escutador()`, para retornar ao comportamento original do ator.

2.11.2 Código do Servidor

Reproduzimos abaixo o código do servidor, e, da mesma forma como fizemos com o código do cliente, comentaremos seus aspectos mais importantes em seguida.

```
1 import paweb
2 import pickle
3 import random
4 from paweb.log import say
5
6 participantes = []
7
8 def main():
9     print("Sou o servidor. Iniciando.")
10    paweb.log.init("server.log",True)
11    paweb.init("server", "", 6543)
12    rec = paweb.actor(recepcionista, [])
13    print("Iniciando servico...")
14    paweb.start_service(rec)
15    print("Pronto.")
16    paweb.run()
17
18 def recepcionista(me):
19    print("Esperando mensagens.")
20    me.receive( \
21        ("OI", paweb.bindvar("ator"), paweb.bindvar("nome")),
              cadastra, [],
22        ("MSG", paweb.bindvar("quem"), paweb.bindvar("msg")),
              fala, [],
23        paweb.bindvar("lixo"),
              erro, [])
24
25 def cadastra(me, ator, nome):
26    global participantes
27    print("Novo participante: %s" % (nome))
28    participantes.append(ator)
29    me.become(recepcionista)
30
31 def fala(me, quem, msg):
32    global participantes
```

```
33      print("%s falou '%s'. Repassando." % (quem,msg))
34      for p in participantes:
35          paweb.send(p, ("MSG", quem, msg))
36      print("Fala enviada a todos os participantes.")
37      me.become(recepcionista)
38
39  def erro(me, lixo):
40      print("ERRO: mensagem inesperada --> %s" % (repr(lixo)))
41      me.become(recepcionista)
42
43  main()
```

O código começa pela inicialização do servidor, especificando endereço e porta para a abertura do *socket* que receberá as conexões dos clientes (linha 11). Em seguida, o servidor cria o ator de recepção, armazenando sua referência na variável `rec`, linha 12. A função que define o comportamento desse ator é a função `recepcionista()`. Uma vez criado o ator recepcionista, o código inicia o serviço com uma chamada a `paweb.start_service` (linha 14), que faz com que o servidor passe a aceitar conexões de clientes. Em seguida, a chamada a `paweb.run()` coloca o sistema em execução.

A função `recepcionista()` define o comportamento do ator recepcionista, que é o único ator do servidor. No nosso exemplo, esse ator recebe mensagens de clientes, que podem ser de dois tipos:

1. *Handshake*. Nesse caso a mensagem é uma tupla Python cujo primeiro elemento é a string literal "OI", e a mensagem contém o endereço do ator do cliente que receberá as mensagens (o ator "escutador") e o nome do usuário.

2. Mensagem. Nesse caso, a string literal da tupla será "MSG" e a mensagem indica que um cliente deseja enviar uma fala a todos os demais.

O comportamento para cada tipo de mensagem é definido pelas funções `cadastra()` (para *handshakes*) e `fala()` (para mensagens que contêm falas dos clientes a transmitir aos demais). A função `cadastra()` simplesmente adiciona o cliente (e o endereço do seu ator "escutador") a uma lista (`participantes`). Por outro lado, a função `fala()` repassa uma fala enviada por um participante a todos os demais participantes.

Esse exemplo também ilustra a transparência de envio de mensagens. Observe que, na linha 36, todos os envios são remotos, ou seja, são mensagens

que se originam no servidor e são entregues aos clientes, mas a sintaxe é idêntica à que seria usada para envios locais de mensagens entre atores na mesma máquina.

2.12 Conclusão

Neste capítulo, descrevemos o funcionamento da plataforma do ponto de vista do usuário, isto é, do desenvolvedor que escreve o código hospedado. Apresentamos a arquitetura do sistema de modo geral, discutimos a representação de atores no sistema e explicamos como efetuar as operações básicas como envio de mensagens, recebimento de mensagens com casamento de padrões, inicialização do sistema, endereçamento de atores e chamadas de cauda. Também apresentamos um exemplo de código consistindo num aplicativo completo de bate-papo, que ilustra a aplicação dos conceitos apresentados. No capítulo seguinte, apresentaremos o funcionamento interno da plataforma, isto é, as técnicas e estratégias utilizadas para possibilitar a implementação da API apresentada nesse capítulo.

Capítulo 3

Implementação

Neste capítulo, discutiremos a implementação da infraestrutura do PAWEB em detalhe, apresentando as técnicas que utilizamos para fornecer a plataforma de programação e API apresentada no capítulo anterior.

3.1 Desafios

Como descreveremos a seguir, o ambiente *web*, embora seja muito versátil, não fornece uma infraestrutura imediatamente adequada para a implementação do modelo de atores. Nesta seção, descreveremos alguns dos principais desafios e qual a combinação de técnicas utilizadas para contornar esses desafios na implementação do PAWEB.

3.1.1 Escolha do Protocolo

Em princípio, como o nosso trabalho define uma nova plataforma, teríamos a liberdade de projetar um protocolo próprio sobre o TCP/IP para transportar as mensagens conforme necessitássemos. Porém, para tornar o sistema mais flexível e compatível com o ambiente existente atualmente na *web*, optamos por implementar o protocolo sobre a infraestrutura existente, utilizando para tal o protocolo HTTP [FGM+99].

3.1.2 Compatibilidade com Navegadores

Os navegadores *web* modernos são extensíveis, isto é, é possível escrever componentes que implementam novas funcionalidades, componentes esses que podem ser instalados e desinstalados facilmente pelo usuário. Esses componentes são denominados *plugins*.

Para que o sistema funcionasse diretamente no navegador do usuário sem a necessidade da instalação de um novo programa, escolhemos implementar o componente cliente do PAWEB como um *plugin* por meio da interface de programação NPAPI, que define uma série de funções que o navegador expõe para permitir que sua funcionalidade seja estendida [Mozc]. Os navegadores da família Mozilla Netscape são compatíveis com essa interface [Net].

3.1.3 Protocolo HTTP

Um dos protocolos mais utilizados na comunicação entre clientes e servidores na *web* é o *Hypertext Transfer Protocol - HTTP*, que, como explicam Fielding et al. [FGM+99], tem a característica de ser sem-estado ("*stateless*"). Isto significa que, ao menos no nível do protocolo, o servidor não armazena nenhuma informação persistente sobre o cliente após o término de cada requisição. O resultado desta característica, como explica Krishnamurthi [Kri07, 148], é que a aplicação Web fica responsável por implementar seus próprios mecanismos para restaurar o estado da computação a cada interação.

O protocolo HTTP [FGM+99] baseia-se, entre outros, nesse princípio de interação, pois essa é uma maneira bastante adequada para fornecer conteúdo relativamente estático. Esse era praticamente a única função da *World Wide Web* nos seus primórdios e ainda hoje continua sendo a função principal de muitos servidores.

Uma das maneiras encontradas pelos projetistas de sistemas para contornar essa deficiência foi a utilização de *"cookies"*. Como explica Kristol [Kri01], os *cookies* eram uma especificação originalmente elaborada pela Netscape [Net] mas que tornaram-se um padrão da IETF [ISO] e conquistaram a indústria.

Os *cookies* são pequenas unidades de informação que o cliente fornece ao servidor a cada vez que inicia uma interação, de forma que o servidor possa identificá-lo e correlacioná-lo com interações anteriores. Por meio do uso de *cookies*, é possível implementar aplicativos razoavelmente complexos apesar das limitações do HTTP. Porém, como adverte Kristol [Kri01], essa funcionalidade também se tornou o centro de uma grande controvérsia envolvendo privacidade e coleta de informações de usuário, justamente por colocar no protocolo HTTP uma maneira de identificar clientes em conexões subsequentes.

O PAWEB não utiliza *cookies* propriamente ditos, pois implementamos o protocolo diretamente sobre o HTTP, e portanto podemos definir nossas próprias maneiras mais apropriadas para identificar a máquina cliente. Em particular, como explicaremos posteriormente, atribuímos a cada cliente um

"identificador de máquina virtual" que o complemento envia juntamente com as requisiões ao servidor. Não podemos deixar de mencionar que esse mecanismo é, em essência, muito similar à função de um *cookie* e de fato foi inspirado nesse conceito. Porém é implementado pelo complemento ao invés de deixarmos a identificação a cargo do navegador.

3.1.4 Falta de Estado

A falta de "estado" do protocolo HTTP é um desafios na implementação do PAWEB porque a existência de atores de ambos os lados da conexão e a interação entre eles pressupõe a manutenção de um estado persistente. Isso significa que o servidor deve continuar alerta quanto à existência do cliente e aos atores existentes no cliente, mesmo nos intervalos em que o cliente está completamente desconectado do servidor.

Convém lembrar que o método ingênuo de utilizar o endereço IP da máquina cliente para identificá-lo não é uma solução razoável, já várias instâncias do PAWEB podem estar ativas na mesma máquina (e portanto com o mesmo IP). Também pode ocorrer a situação na qual uma série de máquinas compartilham o mesmo endereço IP porque estão localizadas numa sub-rede atrás de um *firewall*.

Para contornar esses problemas, o servidor mantém uma "sessão" ativa para cada novo cliente que se conecta, e atribui ao cliente um identificador único de sessão. A partir daí, cada requisição do cliente deve incluir esse identificador, para que o servidor possa direcioná-la para a "sessão" apropriada. Quando o cliente se desconecta, ou quando um intervalo grande de tempo transcorre entre uma requisição e outra, a sessão é removida do servidor com a limpeza adequada, isto é, com a notificação do aplicativo-hóspede de que um cliente se desconectou da aplicação.

Este é o mesmo método utilizado pela maioria das linguagens de programação modernas voltadas ao desenvolvimento de aplicativos *web*, como por exemplo PHP [PG], ASP.NET [Mic98], entre outras.

3.1.5 Modelo "Pull"

Normalmente, num aplicativo *web*, a comunicação entre um navegador e um servidor se dá segundo o paradigma *pull*, isto é, a máquina cliente é responsável por iniciar a interação com o servidor sempre que necessita trocar informações, enquanto que o servidor se limita a responder às requisições do

cliente. No modelo *pull*, o servidor nunca iniciará uma conexão com o cliente sem uma requisição. Essa é uma característica fundamental do protocolo HTTP [FGM+99].

Essa característica é um desafio significativo a uma implementação distribuída do modelo de atores, pois o modelo pressupõe que só o cliente pode iniciar interações com o servidor. Uma das razões para essa decisão de projeto é o fato de que, como explica Krishnamurthi [Kri07, 148], caso as conexões fossem duradouras e bidirecionais, o número máximo clientes que poderiam ser atendidos seria limitado pelo número máximo de conexões simultâneas que o servidor pode manter. Assim, a vantagem dessa característica do protocolo é que o servidor pode lidar com um número de clientes muito maior do que sua capacidade de conexão.

Porém, nas aplicações *web* modernas, esse argumento está perdendo seu efeito porque, como explica Russell [Rus06], muitas delas já utilizam métodos (como por exemplo o Comet) que permitem contornar essa característica do protocolo e desta forma manter conexões duradouras com o servidor. Nessas aplicações, a vantagem da ausência de estado apontada por Krishnamurthi [Kri07, 148] desaparece por completo, já que cada cliente ocupará, no mínimo, uma das conexões disponíveis do servidor.

O modelo *"pull"* do HTTP é particularmente inadequado para a implementação de um modelo de atores porque os atores do servidor poderão, a qualquer momento, enviar mensagens aos atores residentes do lado do cliente. Isto pode ocorrer sem que o cliente esteja esperando por tal interação. Uma possível solução seria implementar um *"blocking-send"*, isto é, colocar na especificação da plataforma que a operação de envio de mensagens pode resultar no bloqueio do ator até que a mensagem seja enviada. Isto resolveria o problema porque, no caso de mensagens do servidor para o cliente, o ator que deseja enviar a mensagem poderia permanecer bloqueado até que o cliente fizesse uma requisição ao servidor, e nesse momento o cliente receberia a mensagem e o ator seria liberado para continuar sua execução.

Porém, acreditamos que esse modelo traz consigo uma série de desvantagens, uma das principais sendo que o ator do servidor desperdiçaria um tempo considerável aguardando o cliente receber as mensagens, tempo esse que poderia ser gasto em outras tarefas. Isto é especialmente grave no caso de um dos clientes, por motivo de lentidão ou de erro de programação, nunca iniciar uma nova requisição ao servidor, fazendo com que o ator no servidor permaneça bloqueado indefinidamente.

Outra solução seria o *polling*, isto é, fazer com que o cliente se conecte ao servidor periodicamente para verificar se existem ou não novas mensagens a enviar. Porém, essa abordagem apresenta um grave problema de latência e as inúmeras conexões e desconexões provenientes de vários clientes simultâneos podem prejudar o desempenho do servidor.

Assim, acreditamos que uma solução melhor para o problema é utilizar um modelo de "simulação de *push*" denominado Comet, que, como explica Russell [Rus06], consiste em substituir o conceito de *polling* (conexões repetitivas do cliente ao servidor para testar se existem dados a receber) por uma "requisição HTTP duradoura." Outros autores também utilizam o termo "Ajax Reverso", como por exemplo Crane [CM08], pois a técnica consiste numa reversão dos papéis costumeiros do cliente e do servidor no modelo "Ajax."

O modelo Comet consiste em deixar sempre uma requisição do cliente "pendente" no servidor, isto é, a todo momento existe uma requisição HTTP que foi feita pelo cliente mas ainda não foi atendida pelo servidor. Sempre que o servidor precisa se comunicar com o cliente, consegue fazê-lo imediatamente respondendo a essa requisição pendente. A requisição se extingue após respondida, como prescreve o protocolo, mas no momento em que o cliente recebe a resposta, ele já faz imediatamente uma nova requisição para substituir a antiga.

Vale lembrar que no HTTP 1.1 [FGM+99] existe também a possibilidade de utilizar o cabeçalho *"Connection"* para tornar esse processo mais ágil. Esse cabeçalho permite realizar várias requisições dentro da mesma conexão TCP/IP, evitando assim os atrasos normalmente incorridos ao desfazer e refazer a conexão a cada requisição. Esta é uma vantagem significativa em relação ao HTTP 1.0, versão do protocolo na qual era necessária uma nova conexão a cada requisição. Dessa forma, a comunicação entre o servidor e o cliente torna-se quase simétrica: sempre que uma das partes deseja enviar uma mensagem para a outra, existe um método quase imediato que se presta a fazê-lo, sem a necessidade de aguardar a outra parte iniciar a interação.

Há vários arcabouços e plataformas que utilizam o método Comet em sua infraestrutura. Por exemplo, o arcabouço Lift [Pol], que se integra com a linguagem Scala, utiliza esse método para comunicar modificações sucessivas nos componentes da aplicação [PV10].

3.1.6 Implementação de Atores

Uma vez que atores são unidades autônomas executadas concorrentemente, uma das implementações mais naturais de atores numa plataforma moderna é a utilização de de "processos" ou de *"threads"*. Ambos são estruturas que permitem a execução concorrente de código, mas, como explica Tanenbaum [TW97, 53-54], *threads* fornecem múltiplos caminhos de execução simultâneos dentro do mesmo processo, com memória compartilhada e com o mesmo espaço de endereçamento. Por outro lado, processos têm espaços de endereçamento diferentes e não compartilham memória entre si.

Como explicado anteriormente, parte do conceito fundamental de atores é que toda a comunicação é feita pela troca de mensagens e nunca pelo compartilhamento direto de memória. Logo, em teoria, processos e threads são igualmente adequados para a implementação de atores, uma vez que o compartilhamento de memória não é utilizado de uma forma ou de outra.

Convém observar, no entanto, que a utilização de *threads* pode proporcionar desempenho superior em certas implementações porque, de maneira geral, sua criação e escalonamento são mais leves do que a criação e escalonamento de processos. Isto ocorre justamente por causa da coincidência do espaço de endereçamento, o sistema operacional não necessita modificar o mapeamento de memória [TW97, 56].

Ainda que as *threads* sejam mais leves que processos (de fato, também são chamadas de *light-weight processes* [TW97, 54]), sua criação e gerenciamento pode causar um impacto significativo no desempenho do programa. Isso ocorre especialmente se for utilizado um grande número delas, afinal há uma variedade de estruturas de dados e outros detalhes de infraestrutura que precisam ser mantidas pelo sistema operacional para implementá-las [TW97, 56]. Por esse motivo, certas linguagens e plataformas, notoriamente o Erlang [Eri], primam-se pela capacidade de criar, gerenciar e escalonar a execução de atores internamente sem a necessidade de utilizar uma *thread* para cada um. Ao invés disso, a plataforma gerencia e escalona esses atores utilizando um número reduzido de threads. Desta forma, como explica Armstrong [Arm07], é possível criar dezenas de milhares de atores sem incorrer no impacto (atualmente proibitivo) da criação de dezenas de milhares de *threads*. O nome comumente atribuido a esse tipo de *thread* mais leve é *micro-thread* ou *green thread*, como foram chamadas quando foram implementadas na máquina virtual da linguagem Java [SUN].

Seria desejável utilizar, na implementação do PAWEB, o conceito de *green*

threads para implementar os atores. Porém, na linguagem Python não há suporte para esse tipo de funcionalidade. Por isso, ao invés de utilizar a linguagem Python pura, decidimos utilizar a variante do Python denominada Stackless Python [Chra], pois nessa variante existe o conceito de *tasklet*, que fundamentalmente consiste numa implementação do conceito de uma *green-thread*.

3.1.7 Comunicação entre Clientes

Um dos pressupostos do modelo de atores é que um ator pode se comunicar com qualquer outro ator cujo endereço possua. Uma vez que o PAWEB fornece transparência de localização, todo ator deve ser capaz de enviar uma mensagem a qualquer outro ator não importando em qual máquina o destinatário está localizado. Isto gera um desafio caso o remetente da mensagem esteja localizado numa máquina cliente e o destinatário esteja localizado em outra máquina cliente, já que o protocolo HTTP não permite a comunicação direta entre clientes.

Para superar essa dificuldade, implementamos no servidor do PAWEB um sistema de encaminhamento de mensagens de forma que clientes possam mandar mensagens a outros clientes de maneira transparente. Do ponto de vista dos clientes, todo o processo ocorre exatamente como se estivessem se comunicando diretamente entre si.

3.1.8 Inexistência de Chamadas de Cauda

Como explicamos anteriormente, uma chamada de cauda (*tail call*) ocorre quando a última instrução executada por uma função é uma chamada a outra função. Certos compiladores são capazes de gerar instruções para este caso especial na forma de um salto (*jump*), garantindo que essa chamada não consumirá espaço de pilha [Arm07, 157].

Embora a chamada de cauda não seja um elemento imprescindível para a implementação do modelo de atores, seu uso é muito conveniente e esse tipo de construção é muito usado na linguagem Erlang. Por esse motivo, implementamos um mecanismo especial para possibilitar a realização de chamadas de cauda em Python utilizando primitivas do Stackless Python [Chra], que explicaremos posteriormente.

3.1.9 Envio Assíncrono de Mensagens

Optamos por implementar no PAWEB o envio assíncrono de mensagens entre atores, isto é, quando um ator envia uma mensagem a outro ator, sua execução continua imediatamente, ao invés de ser bloqueada até que o outro ator receba a mensagem. As mensagens enviadas a cada ator permanecem armazenadas numa "caixa de correio" até que sejam recebidas pelo ator destinatário.

Esta escolha resultou num desafio de implementação porque, no Stackless Python, a comunicação entre processos é síncrona e realizada por meio de objetos chamados canais (*channels*). Quando um processo, ou *tasklet*, como descreve Tismer [Tis00], envia uma mensagem por um canal, ficará bloqueado até que um outro processo leia a mensagem daquele mesmo canal. Devido a esse comportamento do Stackless, foi necessário implementar um mecanismo para possibilitar o envio assíncrono de mensagens utilizando dois *tasklets* para representar cada ator, conforme descreveremos na próxima seção.

3.1.10 Recepção Seletiva

No Stackless Python, a recepção de mensagens não é seletiva, isto é, não permite que o processo que está recebendo especifique qual mensagem, dentre as várias disponíveis, deseja receber. Isto é fundamentalmente diferente do comportamento da primitiva `receive` da linguagem Erlang, que possibilita a recepção seletiva de mensagens por casamento de padrões [Arm07, 153].

Para possibilitar um mecanismo semelhante no PAWEB, implementamos um sistema de casamento de padrões compatível com estruturas de dados da linguagem Python, além de um mecanismo por meio do qual mensagens recebidas são armazenadas numa "caixa de correio" e são entregues ao ator solicitante de acordo com os padrões por ele especificados.

3.2 Implementação dos Atores

No PAWEB, decidimos implementar atores por meio de *tasklets* do Stackless Python. O escalonamento e execução dos atores, portanto, é gerenciado por essa plataforma. A comunicação entre atores, no entanto, exige um cuidado maior. No Stackless Python, a comunicação entre *tasklets* pode ser feita de várias formas, incluindo a utilização de memória compartilhada. Porém, no PAWEB, a comunicação está restrita à troca de mensagens, decisão essa

que foi norteada pela necessidade de transparência entre a comunicação local e remota.

Cada ator no PAWEB é um objeto Python que contém os seguintes componentes:

1. O endereço globalmente único do ator
2. A "caixa de correio", que é uma lista que contém as mensagens recebidas pelo ator mas que ainda não foram processadas
3. Uma *tasklet* que representa o ator
4. Uma *tasklet* que representa o ator *proxy*, que explicaremos na seção seguinte.
5. Um *channel* para comunicação interna entre o ator e o *proxy*
6. Um *channel* para comunicação externa

Na seção a seguir, explicaremos a necessidade e o funcionamento das *tasklets* e *channels* que fazem parte da estrutura interna do ator.

3.3 Transmissão Local de Mensagens

Para implementar a transmissão local de mensagens entre atores, pareceu-nos que a maneira mais apropriada no Stackless Python seria empregar a primitiva denominada "canal" (*channel*) para transmitir as mensagens entre um ator e outro. Um canal é uma estrutura que permite que duas *tasklets* troquem mensagens, mensagens essas que podem ser qualquer objeto Python. A comunicação é síncrona, isto é, quando uma *tasklet* envia uma mensagem num canal no qual não há uma *tasklet* aguardando uma mensagem, ocorre um bloqueio. O mesmo ocorre quando uma *tasklet* tenta receber de um canal ao qual não há uma *tasklet* enviando uma mensagem. [Chrb].

Assim, fica claro que uma das primeiras providências que devemos tomar é a implementação de uma fila de mensagens para que os atores possam enviar mensagens assincronamente. Na prática, isso significa que a plataforma fará com que cada ator tenha uma fila que desempenha o mesmo papel da "caixa de correio" da linguagem Erlang, como descrito por Armstrong [Arm07]. Quando um ator envia uma mensagem para outro ator, essa mensagem é imediatamente colocada na fila de mensagens do ator recipiente e o ator que enviou a

mensagem procede normalmente com sua execução, sem bloqueio. Conforme o ator destinatário recebe e processa mensagens, elas são retiradas da sua fila até que a fila fique vazia. Quando a fila está vazia, uma tentativa de receber uma mensagem fará o ator permanecer bloqueado até que uma mensagem esteja disponível.

A implementação da fila de mensagens foi feita como ilustrado na Figura 3.1. Quando um ator externo envia uma mensagem (1), o canal que o ator utiliza é o chamado *canal proxy*. O *ator proxy* então recebe a mensagem por meio desse canal (2) e armazena-a na fila de mensagens (que é, nos termos do Erlang, o *mailbox* do ator). O ator remetente não ficará bloqueado ao enviá-la para o *canal proxy* porque o *ator proxy* constantemente recebe mensagens nesse canal, excetuado o breve período que ele necessita para armazenar uma mensagem na fila.

Quando o ator real deseja receber uma mensagem, ele envia ao *canal proxy* uma mensagem especial chamada *requisição* (3), que, ao ser recebida pelo *ator proxy*, informa-o sobre quais tipos de mensagens o ator deseja receber. Isto é realizado com casamento de padrões, como será explicado a seguir. Logo após enviar a requisição, o ator real inicia a recepção de uma mensagem no *canal interno*, que fará com que ele permaneça bloqueado até uma mensagem ser enviada nesse canal.

Uma vez que o *ator proxy* tem o conhecimento sobre quais tipos de mensagem o ator real está aguardando, ele pode verificar se existe na fila uma mensagem satisfatória, e, em caso afirmativo, removerá a mensagem da fila e escrevê-la-á no *canal interno* (4), pelo qual ela será recebida pelo ator real, liberando então a execução desse ator. Caso não haja uma mensagem adequada na fila, o *ator proxy* voltará a aguardar uma mensagem no *canal proxy* e, a cada vez que receber uma nova mensagem do mundo exterior, verificará se ela casa com algum padrão esperado pelo ator, e, em caso afirmativo, enviá-la-á por meio do *canal interno*. Um detalhe importante a observar é que, como no Erlang [Eri], o fato de uma mensagem não casar com nenhum dos padrões esperados num dado momento não significa que a mensagem será descartada. Ao invés disso, a mensagem é mantida na fila até que o ator a receba.

Essa organização nos fornece a garantia de que o *ator proxy* nunca ficará bloqueado salvo na recepção de mensagens no *canal proxy*. Afinal, a única outra possibilidade de bloqueio seria o momento no qual ele escreve a mensagem no canal interno, mas nesse momento temos a certeza de que o ator real já está bloqueado efetuando uma recepção de mensagem nesse canal, e portanto o *ator*

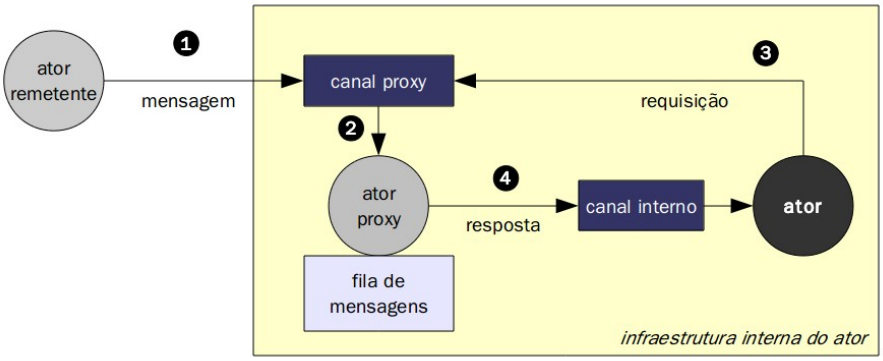

Figura 3.1: *Infraestrutura para Recepção de Mensagens*

proxy não ficará bloqueado ao enviar a mensagem.

Finalmente, no caso de bloqueio do ator *proxy* ao ler do canal *proxy*, temos a certeza de que ou o ator real não está aguardando uma resposta do *proxy* ou está bloqueado aguardando uma mensagem que ainda não está na fila de recepção. Afinal, caso a mensagem existisse na fila, o ator *proxy* teria sido desbloqueado ao receber a requisição de casamento de padrões e teria enviado a mensagem ao ator real. Como a mensagem não existia, o ator *proxy* foi bloqueado no canal *proxy* e aguarda mensagens do mundo exterior.

Expressando o comportamento do *ator proxy* em pseudo-código, teríamos:

1. $B \longleftarrow 0$.
2. $P \longleftarrow \emptyset$
3. Recebe a mensagem M do *canal proxy*.
4. Se M é uma *requisição* do *ator real*,
5. $\quad B \longleftarrow 1$.
6. $\quad P \longleftarrow$ lista de padrões requisitados.
7. Caso contrário, coloca M na fila de mensagens.
8. Se $B \neq 0$ e há na fila uma mensagem N que casa com um padrão $p \in P$.
9. $\quad$ Remove N da fila.
10. $\quad$ Escreve N e as variáveis recém vinculadas no *canal interno*.
11. Volta para o passo 1.

A variável B indica se o ator real está bloqueado aguardando um casamento de padrões ou não, enquanto que P guarda o conjunto dos padrões esperados pelo ator (o conteúdo de P só é relevante caso $B \neq 0$). Desta forma, estamos preparados para processar dois tipos de mensagem que chegam por meio do *canal proxy*:

- *Requisições* oriundas do ator real, que indicam que ele ficará bloqueado aguardando o casamento de algum padrão; e
- Mensagens recebidas do mundo exterior, que são colocadas na fila.

Sempre que recebemos uma mensagem do *canal proxy*, e depois de possivelmente armazená-la na fila, verificamos se o ator real está bloqueado (condição $B \neq 0$) e se existe na fila uma mensagem que casa com algum padrão esperado. Caso haja, essa mensagem é enviada para o ator real juntamente com os demais dados necessários (variáveis vinculadas, entre outros) e portanto o ator real é desbloqueado (e representamos esse fato atribuindo o valor 0 à nossa variável B).

3.4 Endereçamento de Atores

Para que um ator envie uma mensagem a outro ator, é necessário que o remetente possua alguma forma de especificar qual ator é o destinatário da mensagem. Para realizar essa tarefa, a maioria das plataformas utiliza um valor escalar que funciona como *identificador de ator*, também chamado *endereço do ator*.

Uma das características do modelo de atores, como explicamos anteriormente, é que o sistema tem topologia variável conforme os atores são criados e extintos. Uma das características úteis desse tipo de ambiente é a *localidade*, isto é, o fato de que os atores só podem se comunicar com os atores cujos endereços eles já possuam. Naturalmente, os atores podem trocar endereços conhecidos entre si normalmente por meio de mensagens. Porém, na maioria dos sistemas, não costuma ser possível *sintetizar* um endereço de um ator, isto é, construí-lo diretamente sem que ele se origine do próprio ator referenciado ou do conhecimento de outros atores a respeito dele. No caso do Erlang [Eri], por exemplo, os endereços de atores são chamados *identificadores de processo* e são de um tipo opaco para o programador. Em outras palavras, existe a garantia de que o identificador pode ser atribuído a variáveis e manipulado como

um valor qualquer, mas não há como construir arbitrariamente um valor desse tipo.

Decidimos adotar uma filosofia similar no PAWEB. Poderíamos, a princípio, utilizar a própria referência Python para o *canal proxy*, ilustrado na Figura 3.1, como "endereço do ator." Porém, isto só seria adequado caso os atores existissem somente na máquina local. Mas um dos objetivos do PAWEB é justamente fornecer uma infraestrutura transparente para a comunicação entre atores que estão no servidor e atores que estão no cliente. Por isso, quando um ator envia uma mensagem para outro, é necessário que a plataforma intervenha para garantir essa transparência.

Assim sendo, a implementação que escolhemos é fazer com que plataforma mantenha um dicionário com os identificadores (endereços) de ator e alguns dados relacionados com o ator. No caso de um ator local, por exemplo, esse dado seria a referência Python ao canal Stackless que está na função de *canal proxy*.

No caso de um "ator remoto" (um ator do cliente em relação ao servidor ou vice-versa, ou mesmo um ator residente em outro cliente), são necessários dados adicionais para que o PAWEB possa se encarregar de realizar a transmissão da mensagem através da rede sobre o protocolo HTTP. Entraremos em maior detalhe sobre o protocolo de rede do PAWEB posteriormente.

3.4.1 Estrutura de um Endereço PAWEB

Na implementação do PAWEB, o endereço de um ator é uma tupla na qual o primeiro elemento é o identificador da máquina virtual e o segundo é um número inteiro atribuído sequencialmente. O identificador da máquina é fixo para o servidor, mas atribuído dinamicamente no momento do *handshaking* no caso de uma máquina virtual cliente. Uma vez que o servidor atribui identificadores de máquina virtual globalmente únicos e cada máquina virtual atribui identificadores de ator como números inteiros sequenciais e localmente únicos, verifica-se a propriedade de que cada endereço de ator é globalmente único.

3.4.2 Diretório Local de Atores

A plataforma mantém em cada máquina virtual um *diretório local de atores*, que é um dicionário que associa o endereço de cada ator existente na máquina local a uma referência ao objeto Python correspondente. A cada vez que um ator é criado, seu endereço e a referência ao objeto que o representa são colo-

cados nesse diretório. Assim, dado um endereço de um ator local, a plataforma sempre consegue localizar a referência ao objeto Python que o representa.

3.5 Recepção de Mensagens

Um dos problemas com que nos deparamos foi a escolha da forma pela qual o programador do aplicativo hospedado deveria especificar o código que responde à chegada de uma mensagem, isto é, como especificar qual será o comportamento do ator quando ocorre a recepção de uma mensagem que casa com um determinado padrão. A maneira direta é simplesmente indicar o nome de uma função existente, como fizemos para a criação do ator. De fato, no PAWEB esse modelo é utilizado tanto para a criação quanto para a recepção de mensagens. Porém, num primeiro momento, isto nos pareceu uma solução muito inflexível quando comparada com as possibilidades fornecidas pelo Erlang e outras linguagens, razão pela qual explicamos a seguir o raciocínio que levou a essa decisão de projeto.

Em Erlang, existem as *funções anônimas*, também chamadas *funs* [Arm07, 52-53], que podem ser passadas como argumento ou devolvidas de outras funções, uma vez que são um tipo de dados de primeira ordem. Essa primitiva corresponde à primitiva `lambda` da linguagem Scheme, que por sua vez proveio do cálculo lambda originalmente desenvolvido por Alonzo Church como um sistema axiomático formal [SJS76, 30].

Como expõem Steele e Sussman [SJS76], essa primitiva é extremamente versátil e serve para implementar uma série de construções tradicionais na forma de aplicações de funções relativamente simples, utilizando apenas `lambda`, condicionais e, raramente, atribuições. A linguagem Python também fornece uma primitiva `lambda` e permite que funções sejam utilizadas como valores de primeira ordem [Pyta], porém a sintaxe para a criação de uma função anônima é muito mais restrita. Particularmente, a primitiva `lambda` disponível no Python só é apropriada para funções simples, isto é, aquelas que tipicamente só calculam um valor baseado em seus argumentos [Pyta]. Afinal, o uso principal dessas funções no Python é a construção de predicados para as funções de ordem superior da linguagem (como por exemplo `filter`, `map`, entre outras).

A listagem a seguir mostra um exemplo da utilização do `lambda`:

```
1 L = [ 15, 12, 182, 11, 213, 29, 120, 7 ]
2 print filter(lambda x: x % 2 == 0, L)
```

A saída desse exemplo é *[12, 182, 120]*, pois a função anônima implementa o predicado "é um número par" que é passado para a função `filter`. Porém, existe uma restrição: o corpo da função anônima especificada pelo `lambda` em Python deve ser uma única expressão. No Erlang ou no C, talvez essa restrição pudesse ser contornada com o "operador de encadeamento de expressões" (também chamado simplesmente de "operador vírgula"), que permite especificar uma série de expressões a serem avaliadas em sequência. Essa série de expressões é, sintaticamente, também uma expressão cujo valor é o valor da última expressão da sequência. Porém, como no Python não existe tal operador nem outro que lhe corresponda, a restrição de uma única expressão inviabiliza o uso desse recurso para especificar o comportamento completo de um ator.

Felizmente a linguagem Python permite a criação de funções como "variáveis locais," ou seja, novas funções podem ser definidas diretamente dentro do código de outras funções, permitindo a criação dinâmica de novas funções. Isto é bastante diferente do modelo da linguagem C, no qual todas as funções devem estar estaticamente definidas. No seguinte exemplo, criamos uma função dinamicamente a partir de outra:

```
1 def double_fun(fun):
2     def fun2(x):
3         fun(x)
4         fun(x)
5     return fun2
6
7 def myfun(x):
8     print x
9
10 d = double_fun(myfun)
11 d(42)
```

A saída desse programa é o número 42 impresso duas vezes, pois a função `double_fun` cria uma função que chama duas vezes a função que lhe é fornecida, portanto `d` é uma função que chama duas vezes a função `myfun` no seu argumento. Observe que o nome `fun2` está definido somente no escopo local da função `double_fun`, e portanto isto praticamente equivale à criação de uma função anônima.

Naturalmente, esse exemplo se torna muito mais simples em Erlang, devido à possibilidade de utilizar funções verdadeiramente anônimas:

```
1 -module(test).
2 -export([start/0]).
3
```

```
4 doublefun(F) -> fun(X) -> F(X), F(X) end .
5 start() ->
6     F2 = doublefun(fun(X) -> io:format(X) end),
7     F2("42").
```

Ou até mesmo, abusando um pouco mais da palavra-chave `fun`:

```
8 -module(test).
9 -export([start/0]).
10
11 start() ->
12     F = (fun(F) -> fun(X) -> F(X),F(X) end end)(fun(X) -> io:
           format(X) end).
13     F("42").
```

Porém, como explicamos, a inexistência de uma construção similar em Python nos levou à conclusão de que, ao especificar o comportamento do ator ao receber uma mensagem, a aplicação deve indicar o nome de uma função existente e estaticamente definida, que pode residir no módulo atual ou em outro módulo. Esse tipo de referência é seriável em Python, isto é, pode ser representada na forma de uma cadeia de *bytes*, e portanto pode ser transmitida entre atores locais e remotos.

A outra possibilidade que consideramos foi utilizar as funções dinâmicas que resultam da devolução da referência a uma função localmente definida (também chamada *closure*). Porém, esse tipo de construção apresenta a significativa restrição de que a referência não pode ser seriada, e portanto essa possibilidade foi descartada na implementação do PAWEB.

Como exemplo da forma final que decidimos para a recepção de mensagens, considere o trecho de código abaixo:

```
1 import paweb
2
3 def waitformsg():
4     print "Aguardando proxima mensagem."
5
6     paweb.receive(
7         ('nome',paweb.bindvar("nome")), proc_nome,
8         ('telefone',paweb.bindvar("telefone")), proc_tel,
9         paweb.bindvar("desconhecido"), proc_erro)
10
11 def proc_nome(nome):
12     print "Recebi um nome: " + nome
13     me.become(waitformsg)
14
```

```
15 def proc_tel(telefone):
16     print "Recebi um telefone: " + telefone
17     me.become(waitformsg)
18
19 def proc_erro(desconhecido):
20     print "Mensagem inesperada: " + desconhecido
21     me.become(waitformsg)
22
23 myact = paweb.spawn(waitformsg)
24 ...
```

Observe a criação de um ator na linha 23 da listagem, na qual utilizamos o nome da função existente `waitformsg` para especificar o código do novo ator. No momento em que efetuamos uma recepção de mensagens por meio de `paweb.receive`, especificamos os padrões de mensagens que aceitamos, como explicado anteriormente, e também as funções que fornecerão a funcionalidade para o tratamento da mensagem.

Uma observação importante é que a função `paweb.receive` termina o código da função, isto é, nenhum código pode existir depois dela. Escolhemos esse comportamento para emular as chamadas "de cauda" do Erlang usando o Stackless, isto é, sem deixar resíduo na pilha de chamadas (*call stack*). Um efeito semelhante de "chamada de cauda" pode ser obtido com uma chamada a `me.become()`, que termina imediatamente a função atual e substitui-a pela execução da função indicada, sem que haja possibilidade de retorno, pois não há preservação da pilha. Isto é feito nas linhas 13, 17 e 21 da listagem anterior. Este é um dos exemplos que ilustram a flexibilidade do Stackless Python, pois a implementação desse tipo de comportamento de chamada não seria possível no Python tradicional.

A título de comparação, podemos mencionar a linguagem Stage de Ayres e Eisenbach [AE09], na qual a recepção de mensagens por parte dos atores tem uma forma fixa, isto é, cada ator é uma classe Python na qual os nomes dos métodos correspondem às mensagens com as quais o ator está apto a lidar. Nas palavras dos autores, definir um método num ator representa a capacidade do ator de aceitar, processar e possivelmente responder a uma mensagem daquele tipo.

Há duas razões principais pela quais não adotamos essa abordagem. Primeiramente, esse método impossibilita a utilização do casamento de padrões, que é uma característica muito útil que não desejávamos suprimir. Em segundo lugar, acreditamos que vincular a interface do ator com a estrutura estática

da classe Python que o implementa cria um obstáculo nos casos em que o ator precisa mudar seu comportamento dinamicamente, isto é, mudar os tipos de mensagens que são aceitas de acordo com a situação.

3.6 Casamento de Padrões

Uma das funcionalidades mais interessantes do Erlang é o sistema de casamento de padrões, que é utilizando em várias construções da linguagem e é especialmente útil para a troca de mensagens entre atores. Como explica Armstrong [Arm07, 41], o casamento de padrões consiste em fornecer um "molde" que especifica o tipo e a topologia do valor esperado numa determinada operação; esse "molde" pode conter nomes de variáveis que se distinguem em dois tipos: vinculadas e livres (respectivamente, *bounded* e *unbounded*). Ainda segundo o autor, as variáveis vinculadas fornecem valores que se tornam parte do padrão, isto é, impõem uma exigência de conteúdo que o valor candidato deve cumprir; já as variáveis livres não fornecem uma exigência, mas indicam que as partes correspondentes do objeto-alvo devem ser armazenadas numa variável com aquele nome quando o casamento ocorrer com sucesso.

Recorrendo novamente ao Erlang [Eri] como exemplo, consideremos o seguinte padrão:

```
{ { nome, Nome }, { telefone, Telefone } }
```

Suponha, para os efeitos dessa explicação, que a variável `Nome` está vinculada e tem o valor `"Joao"` e a variável `Telefone` não está vinculada, ou seja, é uma variável livre. No Erlang, as chaves (`{` e `}`) indicam uma tupla. Nesse caso, portanto, o padrão exige que o valor candidato seja uma tupla de dois itens, cada um dos quais é uma tupla: a primeira consiste no átomo `nome` seguido do valor específico `"Joao"`, que é especificado pela variável vinculada `Nome`. A outra tupla consiste no átomo `telefone`, e o segundo item é a variável livre `Telefone`, que indica que, no momento em que ocorrer o casamento do padrão, essa variável deverá receber o conteúdo presente na posição correspondente do valor contra o qual o padrão casou.

Considere o valor-candidato abaixo.

```
{ { nome, "Pedro" }, { telefone, "1112345678" } }
```

Esse valor não casaria com o padrão, pois o nome `"Pedro"` é diferente do nome `"Joao"` que faz parte do padrão (já que a variável que o especificou é uma variável vinculada). Já o valor abaixo casa com o padrão.

```
{ { nome, "Joao" }, { telefone, "1123456789" } }
```

Nesse caso, após o casamento do padrão, a variável `Telefone` conteria o valor `"1123456789"`.

Por último, e apenas como uma observação, convém lembrar que um "átomo" em Erlang é meramente uma constante, exceto pelo fato de que seus valores são intrínsecos e não precisam ser declarados (todos os identificadores em minúsculas são átomos em Erlang). Esse conceito não está estritamente relacionado com o casamento de padrões, mas é bastante útil, como o exemplo acima demonstra.

A linguagem Python, diferentemente de Erlang, não possui um mecanismo nativo para realizar o casamento de padrões. Poderíamos prosseguir com a implementação sem utilizar o casamento de padrões, pois isso não é exigido para uma implementação do modelo de atores. Como exemplo, citamos a linguagem Stage criada por Ayres et al. [AE09]. Nessa linguagem, o conceito de casamento de padrões não é implementado porque os atores são construídos como classes Python, nas quais a recepção de uma mensagem automaticamente dispara uma chamada ao método com o nome correspondente ao "tipo" da mensagem.

Porém, embora essa abordagem pareça razoável num primeiro momento, pareceu-nos que exigir que as mensagens tenham essa forma específica e, além disso, vincular a organização do código do ator com a recepção de mensagens, seria um pouco inconveniente para o programador. Por exemplo, caso o ator desejasse mudar de comportamento em resposta a uma mensagem, seria necessário que o ator redefinisse os métodos da classe ou de alguma forma "mudasse" de classe Python. Por isso, escolhemos seguir a filosofia do Erlang, segundo a qual o programador especifica os padrões e os comportamentos correspondentes no próprio comando que realiza a recepção de mensagens.

A implementação do casamento de padrões em Python não é muito complexa em razão dos amplos recursos de reflexão fornecidos pela linguagem, isto é, existem operadores especiais na linguagem que permitem inspecionar o tipo e a topologia de objetos Python em tempo de execução. Assim, o "padrão" pode ser um objeto Python que especifica o tipo e a topologia do valor desejado, e há valores especiais embutidos nele que denotam as variáveis livres que devem ser preenchidas. Não há necessidade de lidar especificamente com com as variáveis vinculadas, pois o interpretador automaticamente substituí-las-á por seus valores correspondentes.

Convertendo o exemplo anteriormente exposto ao formato do PAWEB, teríamos:

```
(("nome",Nome),("telefone",paweb.bindvar("Telefone")))
```

Nesse caso, o próprio interpretador já substituiria o valor da variável `Nome` (variável vinculada) no padrão, então, com efeito, o que recebemos é:

```
(("nome","Joao"),("telefone",paweb.bindvar("Telefone")))
```

A função `paweb.bindvar` indica que desejamos vincular uma variável naquela posição do padrão. A implementação deste mecanismo é que a função devolve um objeto de uma classe interna especial chamada `VarBindMarker` que contém o nome da variável a vincular. No momento do casamento do padrão detectamos esse tipo especial e atribuímos o conteúdo àquela variável. Portanto, a operação de recebimento de uma mensagem no PAWEB emula o comportamento Erlang: a aplicação especifica uma série de padrões de mensagens que são aceitas e, para cada padrão, uma função que será invocada se uma mensagem que casa com aquele padrão for recebida.

No caso do PAWEB, as variáveis casadas no padrão (variáveis livres) são passadas para essas funções por meio de "parâmetros nomeados," que são uma funcionalidade bastante conveniente fornecida pelo Python. O casamento de padrões introduz um problema com a utilização de canais do Stackless, pois teoricamente um ator aguardando uma mensagem só deveria ser desbloqueado para execução quando recebesse uma mensagem que casa com um dos padrões esperados. Porém, a implementação padrão dos canais do Stackless fará com que o ator seja desbloqueado no momento do recebimento de qualquer tipo de mensagem.

Porém, como já explicamos na seção anterior, esse problema é resolvido com a utilização de um *ator proxy* e uma fila de mensagens. A *requisição* enviada pelo ator real ao *ator proxy*, marcada com o número (3) na Figura 3.1, contém uma lista de todos os padrões possíveis para casamento, enquanto que a resposta (4) indica, para o ator real, qual dos padrões foi casado e também fornece a lista dos valores das variáveis livres atribuídas no casamento.

3.7 Algoritmo de Casamento de Padrões

O algoritmo de casamento de padrões implementado no PAWEB baseia-se na comparação recursiva de duas estruturas de dados: o *padrão* e o *valor candidato* (ou simplesmente *valor*). Como explicado na seção anterior, o padrão é um objeto Python que representa a topologia dos valores que casam com o

padrão, e pode conter objetos especiais que indicam as variáveis livres cujo valor será atribuído pelo casamento.

Para descrever o algoritmo, utilizaremos os seguintes tipos de dados:

1. Um *escalar* é qualquer dos tipos primitivos escalares da linguagem Python: números, cadeias de caracteres, entre outros.
2. Um *vetor* é uma lista de valores.
3. Uma *tupla* também é uma lista de valores.
4. Um *dicionário* é uma estrutura que mapeia cada elemento de um conjunto de chaves para um respectivo valor.

A diferença entre *vetor* e *tupla* existe na linguagem Python mas não influencia o algoritmo, pois o tratamento para ambos os casos é idêntico. Portanto, sem perda de generalidade, utilizaremos o termo *vetor* para designar essas duas estruturas na nossa descrição a seguir.

3.7.1 Notação

Se L é um vetor, então denotaremos por $\mathcal{L}(L)$ o comprimento do vetor e por L_i com $i = 0, 1, \ldots \mathcal{L}(L) - 1$ os elementos desse vetor. Se D é um dicionário, então denotaremos por $\mathcal{K}(D)$ o conjunto das chaves desse dicionário. Se $k \in \mathcal{K}(D)$, então D_k é o valor ao qual o dicionário D associa a chave k. Adicionalmente, se D e E são dicionários, então $D \oplus E$ é o dicionário formado pela união dos dicionários, definida da seguinte forma:

Seja $F = D \oplus E$. Então, $\mathcal{K}(F) = \mathcal{K}(D) \cup \mathcal{K}(E)$ e, para cada $k \in \mathcal{K}(D) \cup \mathcal{K}(E)$, $F_k = E_k$ se $k \in \mathcal{K}(E)$, caso contrário $F_k = D_k$.

3.7.2 Algoritmo

Realizar um *teste de casamento de padrão* significa comparar um padrão P a um valor candidato V e decidir se houve ou não casamento e, em caso afirmativo, determinar um dicionário D que relaciona cada variável livre ao seu valor correspondente. O algoritmo implementado pode ser descrito pela função recursiva $\mathcal{C}$ que descrevemos a seguir.

Se P é um padrão e V é um valor candidato, então $\mathcal{C}(P, V)$ devolve uma tupla (m, D) onde $m = 1$ se e somente se houve casamento do padrão e D é o dicionário de variáveis vinculadas resultante do casamento.

$\mathcal{C}(P, V)$ está definida abaixo:

1. Se P é um escalar, então:
2. Se V é um escalar e $P = V$, devolve $(1, \emptyset)$
3. Caso contrário, devolve $(0, \emptyset)$
4. Se P é um vetor, então:
5. Se V não é um vetor, devolve $(0, \emptyset)$
6. Se $\mathcal{L}(V) \neq \mathcal{L}(P)$, devolve $(0, \emptyset)$
7. $D \longleftarrow \emptyset$ (D é um dicionário)
8. Para cada $i = 0 \ldots \mathcal{L}(P) - 1$,
9. $m, E \longleftarrow \mathcal{C}(P_i, V_i)$
10. Se $m = 0$, devolve $(0, \emptyset)$
11. $D \longleftarrow D \oplus E$
12. Devolve $(1, D)$
13. Se P é um dicionário, então:
14. Se V não é um dicionário, devolve $(0, \emptyset)$
15. Se $\mathcal{K}(V) \neq \mathcal{K}(P)$, devolve $(0, \emptyset)$
16. $D \longleftarrow \emptyset$ (D é um dicionário)
17. Para cada $k \in \mathcal{K}(V)$,
18. $m, E \longleftarrow \mathcal{C}(P_k, V_k)$
19. Se $m = 0$, devolve $(0, \emptyset)$
20. $D \longleftarrow D \oplus E$
21. Devolve $(1, D)$
22. Se P é uma variável livre, então:
23. $D \longleftarrow \emptyset$ (D é um dicionário)
24. $D_P \longleftarrow V$

25. Devolve $(1, D)$

As variáveis livres são facilmente identificáveis (linha 22) porque o mecanismo de reflexão em Python permite identificar o tipo de uma variável em tempo de execução, e as variáveis livres são sempre do tipo especial interno `paweb.BindVar`. A função `paweb.bindvar()` devolve um objeto deste tipo. Porém, a existência desse mecanismo é um detalhe de implementação da plataforma e é não é exposto ao desenvolvedor do código hospedado.

3.8 Entrega de Mensagens Locais e Remotas

Quando uma mensagem é enviada a um determinado endereço no PAWEB, a infraestrutura é responsável por determinar como entregá-la ao ator destinatário. Para tanto, implementamos um módulo de comunicação com a seguinte estrutura:

1. Q, uma fila de mensagens remotas a enviar
2. C_Q, uma variável de condição para controlar o acesso a Q. Essa variável de condição possui uma trava tipo *mutex* associada X_Q.
3. T_S, uma *thread* dedicada ao envio de mensagens remotas
4. T_R, uma *thread* dedicada ao recebimento de mensagens remotas
5. Um dicionário D que mapeia endereços de atores locais a seus objetos Python correspondentes. Esse dicionário é chamado *diretório de atores*.

Para enviar uma mensagem m, o código que implementa a infraestrutura dos atores executa o seguinte algoritmo:

1. Seja d o destinatário de m
2. Se d é a máquina local,
3. Procura d no diretório local de atores D
4. Efetua a entrega direta
5. Caso contrário (d é remoto):
6. Obtém o *lock* X_Q
7. Coloca m na lista Q

8. Notifica a variável de condição C_Q

9. Libera o *lock* X_Q

A notificação da variável de condição C_Q serve para comunicar à thread T_S que existe uma mensagem remota a enviar. O algoritmo executado pela thread T_S é simples, e consiste no envio de mensagens conforme elas se tornam disponíveis na fila Q, controlada pela variável de condição C_Q:

1. Enquanto perdurar o programa:

2. Obtém *lock* X_Q.

3. Enquanto $Q = \emptyset$,

4. Aguarda na variável de condição C_Q

5. $n \longleftarrow \mathcal{L}(Q)$ (denota o comprimento de Q)

6. $m \longleftarrow Q_0$

7. $Q \longleftarrow (Q_1 \ldots Q_{n-1})$

8. Libera *lock* X_Q

9. Seja d a máquina onde reside o destinatário de m

10. Conecta-se à máquina d

11. Envia mensagem m

Um dos problemas com o algoritmo da maneira como está exibido acima é que caso haja muitas mensagens para a mesma máquina de destino, múltiplas conexões serão abertas e fechadas, uma para cada mensagem, o que é bastante ineficiente. Uma otimização que implementamos foi a de agrupar mensagens na fila por máquina destinatária para evitar esse fenômeno. Nesse caso, na linha 6, ao invés de obter somente a primeira mensagem da fila, obtemos de uma só vez todas as mensagens da fila cuja máquina destinatária seja a mesma da primeira e mandamos todas as mensagens em um único lote:

1. Enquanto perdurar o programa:

2. Obtém *lock* X_Q.

3. Enquanto $Q = \emptyset$,

4. Aguarda na variável de condição C_Q
5. $n \longleftarrow \mathcal{L}(Q)$ (denota o comprimento de Q)
6. $m \longleftarrow Q_0$
7. Seja $M \subset Q$ o conjunto de mensagens que têm a mesma máquina destinatária de m.
8. $Q \longleftarrow Q \setminus M$
9. Libera *lock* X_Q
10. Seja d a máquina onde reside o destinatário de m
11. Conecta-se à máquina d
12. Para cada $m \in M$,
13. Envia m.

A *thread* de recepção de mensagens T_R exibe o comportamento complementar, ou seja, recebe mensagens advindas da rede e entrega cada uma ao seu ator destinatário apropriado:

1. Enquanto perdurar o programa:
2. Aguarda a recepção da próxima mensagem
3. Seja m a mensagem recebida e d o destinatário de m.
4. Se $d \in D$,
5. Seja a o ator associado ao endereço d no dicionário D.
6. Envia a mensagem ao canal *proxy* de a
7. Caso contrário,
8. Imprime erro

Este algoritmo pressupõe que todas as mensagens recebidas da rede são endereçadas a atores locais, uma vez que de outra forma a máquina remetente não teria contactado essa máquina. Porém, com uma sutil modificação, podemos obter um efeito interessante que descreveremos a seguir.

A linha 2 do algoritmo acima é implementada de maneira diferente no cliente e no servidor. No servidor, aguardar a recepção da próxima mensagem significa executar o seguinte procedimento:

1. Aguardar uma nova conexão no *socket*.
2. Ler os cabeçalhos para determinar o remetente.
3. Construir a mensagem a partir da decodificação do conteúdo.
4. Devolver um código de sucesso ao cliente.
5. Fechar a conexão
6. Devolver a mensagem lida.

Por outro lado, no cliente, aguardar a recepção da próxima mensagem consiste no seguinte procedimento:

1. Abrir uma nova conexão ao servidor.
2. Enviar os cabeçalhos.
3. Aguardar a resposta do servidor.
4. Ler os cabeçalhos e conteúdo da resposta
5. Construir a mensagem a partir da decodificação do conteúdo.
6. Fechar a conexão
7. Devolver a mensagem lida.

Uma vez que o cliente desconhece quando o servidor desejará enviar mensagens, a execução do passo 3 faz com que a execução do cliente permaneça suspensa até que o servidor responda à requisição. O servidor só responderá a requisição quando houver uma mensagem a entregar para o cliente.

Este mecanismo garante que cada cliente sempre terá uma conexão pendente junto ao servidor, isto é, uma requisição cuja resposta ainda não respondeu. Esta técnica é muito conhecida e utilizada, e serve para contornar o fato de que o servidor não pode abrir uma conexão com um cliente por sua própria iniciativa, como explicaremos em maior detalhe na seção 3.12.5.

3.9 Comunicação Direta entre Clientes

A implementação da transmissão transparente de mensagens por meio de de endereços dá margem a uma situação interessante, que é um pouco inusitada no contexto de um aplicativo *web* tradicional: a comunicação direta entre dois clientes. Para ilustrar essa possibilidade, suponha que um cliente A crie um ator a e envie o endereço desse ator para o servidor, e posteriormente o servidor envia esse mesmo endereço para um outro cliente B. Nesse caso, é perfeitamente razoável que o cliente B queira enviar uma mensagem ao ator a. O que ocorre nesse caso é que, primeiramente, a plataforma de execução do cliente B observa que o endereço de a é remoto, logo envia a mensagem para o servidor. O servidor, ao receber a mensagem, percebe que o endereço de a é, novamente, remoto (desta vez em relação ao servidor, pois a reside num cliente). Portanto, o servidor procede ao encaminhamento (*forwading*) da mensagem para o cliente A, onde ela será finalmente entregue ao ator destinatário a.

O efeito final deste mecanismo é que atores residentes em clientes diferentes podem se comunicar diretamente entre si como se o servidor não existisse, pois nesse caso o servidor meramente desempenha o papel de "correio" ou *relay*, sem que o aplicativo hóspede tenha que fornecer qualquer tipo de implementação especial para cuidar dessa funcionalidade. Para implementar o *forwarding* no PAWEB, basta introduzir uma ligeira alteração ao algoritmo da *thread* T_R, que é responsável pelo recebimento e entrega de mensagens remotas:

1. Enquanto perdurar o programa:
2. Aguarda a recepção da próxima mensagem
3. Seja m a mensagem recebida e d o destinatário de m.
4. Se d é um endereço local,
5. Se $d \in D$,
6. Seja a o ator associado ao endereço d no dicionário D.
7. Envia a mensagem ao canal *proxy* de a
8. Caso contrário (d é remoto),
9. Obtém o *lock* X_Q

10. Coloca m na lista Q

11. Notifica a variável de condição C_Q

12. Libera o *lock* X_Q

Esta alteração faz com que, caso a máquina receba da rede uma mensagem cuja máquina destinatária não é a máquina atual, a mensagem seja colocada na fila de envio remoto, que resulta no comportamento de *forwarding* que desejamos.

3.10 Implementação das Chamadas de Cauda

Recordemos que as chamadas de cauda (*tail calls*) não são otimizadas em Python, isto é, deixam resíduo na pilha de chamadas, diferente do que ocorre na linguagem Erlang. Como explicamos anteriormente, optamos por implementar no PAWEB uma função que permite a chamada de cauda sem resíduo de pilha, pois essa estrutura é muito útil para programas baseados no modelo de atores. O código hospedado chama o método `become` para efetuar uma chamada desse tipo:

```
1 def fun_1(me):
2    print "Sou a funcao 1"
3    me.become(fun_2)
4
5 def fun_2(me):
6    print "Sou a funcao 2"
```

Para realizar esta implementação sobre a infraestrutura do Stackless Python, a técnica que utilizamos é encerrar a *tasklet* atual e iniciar uma nova *tasklet* com a nova função. Esse efeito é obtido da seguinte maneira:

```
1 def become(self, fun, args=[], kwards={}):
2    self.main_tlet = stackless.tasklet(self, *args, **kwargs)
3    stackless.getcurrent().kill()
```

3.11 Funcionamento do Servidor

3.11.1 O Papel do Servidor

O servidor, PAWEBServer, é um programa em Python que aguarda conexões HTTP [FGM+99] numa porta conforme configurado no seu arquivo de

configuração, gerenciando o aplicativo-hóspede servidor e lidando com as requisições dos clientes em baixo nível. Em outras palavras, o PAWEBServer é responsável por receber as conexões via *sockets* e traduzir as mensagens recebidas por meio do protocolo HTTP para os termos do PAWEB, para que o aplicativo-hóspede as veja como trocas de mensagens entre atores. Essa parte da infraestrutura é muito similar àquela fornecida pelo *plugin* no lado do cliente.

3.11.2 Inicialização do Servidor

Durante a inicialização, o servidor inicia o módulo de comunicação descrito na seção 3.8 e aguarda até que o código hospedado crie o *ator de recepção*, isto é, o ator que será o destinatário da primeira mensagem de clientes que se conectam ao servidor. Uma vez que isto é feito, o código hospedado chama `paweb.start_service`, e o servidor abre um *socket* UNIX em modo *listen* para aguardar conexões de clientes.

O ator recepcionista pode ser referenciado pelo endereço especial constante `RCPT_ADDR`, que é o valor da constante `paweb.RECEPTIONIST`. Este é o único ator no sistema com endereço pré-fixado. Por exemplo, para enviar uma mensagem m a esse ator, o código (tanto no cliente quanto no servidor) poderia utilizar a seguinte sintaxe:

```
1 paweb.send(paweb.RECEPTIONIST, m)
```

3.11.3 Recepção de Conexões

Um cliente se conecta ao servidor fazendo uma requisição HTTP à página-raiz. Por exemplo, se o servidor está localizado numa máquina cujo endereço é `abc123.net` e está configurado para servir na porta 8080, então a primeira requisição que o cliente fará (por meio do navegador) será um HTTP GET [FGM+99] da seguinte URL:

```
http://abc123.net:8080/
```

O PAWEBServer responderá a essa requisição com uma resposta HTTP 1.1 [FGM+99] indicando que a renderização do conteúdo da página deve ser delegado ao complemento de navegador PAWEBPlugin, que é consequência da identificação do tipo do conteúdo como o *MIME type* que elegemos para representar o PAWEB. A estrutura da resposta é:

```
HTTP/1.1 200 OK
Date: <data>
Content-Type: application/x-paweb-client-code
Content-Length: <comprimento>

<conteudo>
```

A resposta conterá todos os dados que o *plugin* necessita para inicializar a execução do código cliente, isto é, todos os módulos Python que compõem o código da parte cliente da aplicação-hóspede e informações de configuração adicionais. Entre essas informações podemos destacar o *identificador de cliente*, que será um número único gerado pelo servidor e que será utilizado para identificar o cliente para fins de endereçamento de atores.

O código do aplicativo hospedado não é notificado automaticamente que um novo cliente se conectou. A notificação é deixada a cargo do código do cliente, que conhece o endereço do ator de recepção. O cliente tipicamente enviará uma mensagem ao servidor para notificá-lo de sua existência, mas é possível que o desenvolvedor prefira fazer com que o cliente aguarde até que o usuário realize alguma ação específica antes de entrar em contato com o servidor. Por exemplo, caso o aplicativo necessite de autenticação, o cliente pode esperar até que o usuário digite o seu nome de usuário e senha antes de enviar a mensagem de *handshaking* ao servidor, para poder incluir a informação da autenticação nessa mensagem inicial.

3.11.4 *Thread* de Trabalho

O servidor inicia uma nova *thread*, chamada *thread de trabalho*, para cada cliente que se conecta ao servidor. Essa thread sobrevive enquanto durar a conexão, e é responsável por se comunicar com aquele cliente específico.

Esta *thread* compreende três tipos de requisição:

1. Registro: o cliente está se identificando com o servidor.
2. Leitura: o cliente solicita suas mensagens pendentes.
3. Envio: o cliente envia mensagens.

Cada um desses tipos de requisição será descrito a seguir.

3.11.5 Requisição de Registro

O cliente envia uma requisição de registro ao servidor no momento em que se conecta pela primeira vez. Esta requisição contém o seguinte cabeçalho:

```
GET /hello HTTP/1.1
Date: <data>
...
```

Em resposta a essa requisição, o servidor gerará um identificador de máquina único para o cliente e responderá com esse identificador e, adicionalmente, com o endereço real do ator de recepção. Convém esclarecer que anteriormente descrevemos o ator de recepção como um ator de endereço *fixo*. Porém, há um detalhe de implementação que é invisível para o código hospedado: o endereço fixo `RCPT_ADDR` é, em realidade, um sinônimo para um endereço real, que é o endereço informado pelo servidor nessa resposta.

O formato da resposta será:

```
HTTP/1.1 200 OK
Date: <data>
Content-Type: text/plain
Content-Length: <comprimento>

<node_id>
<rcpt_addr>
```

onde `<node_id>` representa o identificador da máquina e `<rcpt_addr>` representa o endereço do ator de recepção.

3.11.6 Requisição de Leitura

O cliente envia essa requisição para obter suas mensagens pendentes. Uma mensagem pendente é uma mensagem que existe na fila correspondente ao cliente, fila essa mantida pelo servidor para cada cliente conhecido. Sempre que o servidor recebe uma mensagem endereçada a um cliente C, essa mensagem é armazenada numa fila para o cliente C até que esse cliente se conecte e efetue essa requisição. Esta arquitetura é necessária no lugar de fazer um envio imediato por causa da limitação imposta pelo protocolo HTTP, que impossibilita que o servidor inicie uma conexão com um cliente.

Esta requisição contém um cabeçalho idêntico ao da requisição de registro, mas com `GET /msgs` no lugar de `GET /hello`. Quando o servidor recebe esta mensagem, responderá imediatamente caso haja mensagens pendentes a entregar ao cliente. Porém, caso não haja mensagens, ao invés de desconectar-se ou devolver uma resposta vazia, o servidor manterá a conexão aberta por tempo indeterminado até que haja mensagens a transmitir.

Como explicado anteriormente, este comportamento visa contornar o fato de que o servidor não pode iniciar uma conexão. Desta forma, quando o servidor tiver uma mensagem a enviar para o cliente, poderá fazê-lo respondendo à requisição pendente existente.

O formato da resposta é:

```
HTTP/1.1 200 OK
Content-Type: application/octet-stream
Content-Length: <comprimento>

<conteudo>
```

onde `<conteudo>` é a representação *seriada* (por meio do módulo `pickle` do Python) de um *envelope*. Um envelope é um objeto que contém:

1. o endereço do ator remetente;
2. o endereço do ator destinatário;
3. a mensagem em si.

3.11.7 Requisição de Envio

O cliente envia essa requisição quando deseja enviar mensagens para o servidor ou para outros clientes. Esse tipo de requisição é identificado pelo cabeçalho `POST /msg`. Diferente das mensagens tipo `GET`, essa mensagem contém um corpo, isto é, deve vir acompanhada de um cabeçalho `Content-Type`, `Content-Length` e da mensagem em si, conforme especifica o protocolo [FGM+99].

O formato da requisição é:

```
POST /msg HTTP/1.1
Content-Type: application/octet-stream
Content-Length: <comprimento>

<conteudo>
```

onde <conteudo> representa um envelope *seriado*, da mesma forma como explicado na seção anterior.

Convém observar que normalmente um cliente fará a requisição de envio a partir de uma conexão diferente daquela que está utilizando para enviar requisições de recepção. Isto ocorre porque as requisições de recepção são bloqueadas até que haja uma mensagem endereçada ao cliente, e durante esse tempo o cliente pode desejar enviar uma mensagem a outros atores.

3.12 Funcionamento do Cliente

3.12.1 Inicialização

Como explicamos na seção sobre o funcionamento do servidor, o cliente inicia fazendo uma requisição para a URL da aplicação. Quando o navegador recebe o *MIME type* do PAWEB, delega a execução da página para o *plugin*.

Após a inicialização do *plugin*, o cliente efetuará o *handshaking* com o servidor, que é feito enviando-se uma requisição de registro. O servidor responderá a essa requisição com o ID do cliente e com o endereço do ator de recepção do servidor, ambos os quais serão armazenados para uso posterior.

3.12.2 *Threads* de Envio e Recepção

O cliente então inicia duas *threads*, chamadas T_R e T_S. A thread T_S é a thread de envio de mensagens, isto é, a thread que se encarregará de enviar uma mensagem ao servidor sempre que houver uma mensagem a ser enviada. A thread T_R, por outro lado, é a thread que se encarrega de conectar-se constantemente ao servidor para solicitar mensagens pendentes endereçadas ao cliente, e, conforme as recebe, efetuará a entrega para os atores apropriados.

3.12.3 Fila de Mensagens Remotas

A infraestrutura do cliente mantém uma fila de mensagens Q que armazena todas as mensagens remotas que necessitam ser enviadas ao servidor. Quando um ator deseja enviar uma mensagem remota, seja para o servidor ou para outro cliente, deposita-la-á na fila de mensagens Q, de onde a thread T_S as obterá para enviar. A fila Q está protegida por mecanismos de exclusão de concorrência e também por uma variável de condição associada, que permite

que o ator sinalize quando uma nova mensagem é colocada na fila, fazendo com que a thread T_S desperte para enviá-la.

3.12.4 Execução

A infraestrutura de execução de código no cliente é muito semelhante à do servidor, isto é, fornece as mesmas abstrações de atores e de passagem de mensagens entre atores. Naturalmente, a funcionalidade de envio de mensagens aos "atores remotos" também é fornecida: sempre que o cliente tenta enviar uma mensagem a um ator que reside no lado do servidor, ou em qualquer outra máquina remota, a plataforma traduzirá o envio da mensagem para uma requisição HTTP ao servidor. A sintaxe para envio e recebimento de mensagens é exatamente a mesma, não importando se o ator remetente ou destinatário está no cliente ou no servidor.

3.12.5 Requisição Pendente Constante

Como explicamos anteriormente, um dos desafios do protocolo HTTP é que somente o cliente pode iniciar uma conexão com o servidor, nunca o contrário. Para resolver esse problema, utilizamos uma abordagem bastante conhecida que é a de manter sempre uma requisição "pendente" no servidor. Isto é, o plugin sempre iniciará uma conexão HTTP com o servidor independentemente de haver uma mensagem para enviar ou não, conexão essa que o servidor poderá utilizar para transmitir uma mensagem para o cliente quando desejar.

Portanto, cada cliente PAWEB poderá utilizar até duas conexões com o servidor. Uma delas é a conexão permanente que serve para a comunicação do servidor para o cliente, e a outra é uma conexão intermitente que o PAWEB utiliza somente quando necessita enviar dados.

É oportuno observar que existe uma nova funcionalidade que surgiu com o HTML 5.0, chamada *web sockets*, que foi proposta por Hickson [Hic10] como padrão na IETF [ISO]. Como explica Yu et al. [YNL09], os *web sockets* permitem a comunicação entre um navegador e qualquer *back-end* TCP/IP. Esta funcionalidade seria ideal para o PAWEB, pois, como apontam Kuuskeri et al. [KM09], os *web-sockets* são um canal bidirecional de comunicação entre o servidor e o cliente que torna desnecessário abrir repetidas conexões ou deixar requisições pendentes, afinal qualquer um dos lados pode enviar dados a qualquer momento, dependendo somente do protocolo estabelecido pela própria aplicação. Como expõe Hickson [Hic09], os *web-sockets* fornecem uma API

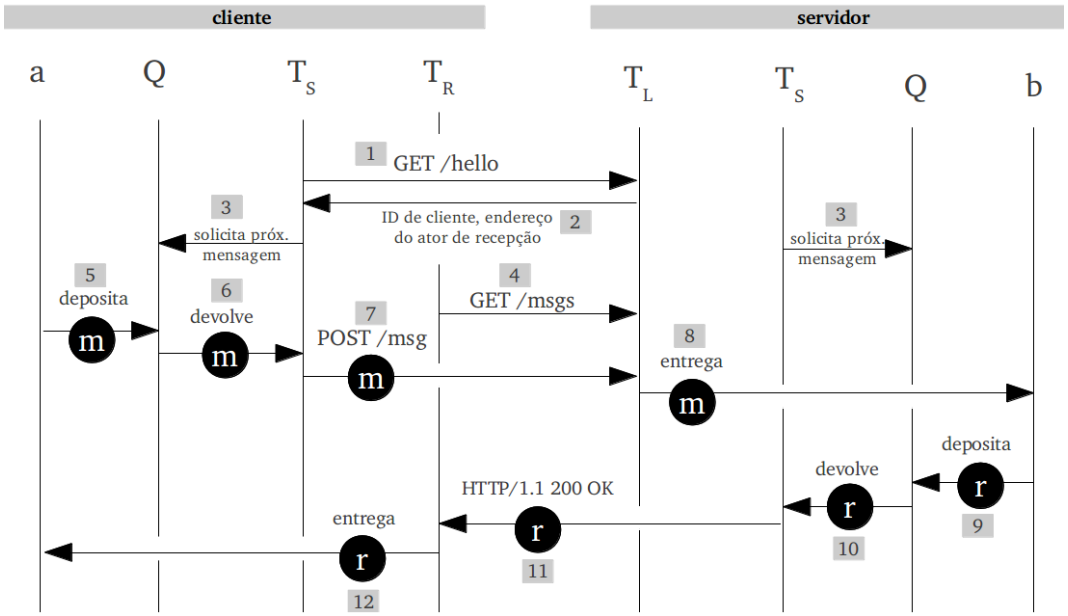

Figura 3.2: *Percurso de uma mensagem e uma resposta*

razoavelmente simples quando comparada aos métodos Comet e Ajax. No entanto, optamos por não utilizar *web-sockets* na implementação do do PAWEB porque ainda não existe uma padronização definitiva da interface e ainda são poucos os navegadores (dentre os quais destaca-se o Google Chrome [Goo]), servidores e *proxies* que dão suporte a essa funcionalidade.

3.13 Percurso Completo de uma Mensagem

Para ilustrar o funcionamento e a interação entre os diversos componentes descritos nas seções anteriores, observemos a Figura 3.2. Este diagrama exibe um exemplo de execução de um servidor e um cliente hipotéticos, no qual o cliente envia uma mensagem m ao ator de recepção do servidor e o ator de recepção responde com uma mensagem r.

Adotamos a seguinte notação:

1. a é um ator que reside no cliente
2. b é o ator de recepção do servidor
3. Q é a fila de mensagens remotas. Existe uma fila Q do lado do cliente e outra do lado do servidor.

4. T_S é a *thread* de envio de mensagens remotas. Essa thread existe tanto do lado do servidor quanto do lado do cliente, porém com comportamentos ligeiramente diferentes, como descreveremos a seguir.

5. T_L é a *thread* do servidor responsável por aguardar conexões e requisições e serví-las. Esta thread é exclusiva do servidor.

6. T_R é a *thread* do cliente responsável por contactar o servidor constantemente para obter novas mensagens.

A inicialização do cliente consiste em enviar uma requisição de registro ao servidor (1), à qual o servidor responde com o ID do cliente e o endereço do ator de recepção (2). Neste caso, o ator de recepção é o ator b.

Em seguida, as threads T_S do servidor e do cliente são bloqueadas na variável de condição de suas respectivas filas de mensagens (Q) aguardando até que haja uma mensagem remota a transmitir. Uma vez que as filas estão vazias, T_S permanece bloqueada (3).

Em seguida, a *thread* T_R do cliente entra em execução e efetua uma requisição de leitura (4), que solicita ao servidor que envie as mensagens pendentes endereçadas ao cliente. Uma vez que ainda não há nenhuma mensagem pendente, o servidor não responde imediatamente. Ao invés disso, conserva a conexão aberta até que haja uma mensagem a enviar. O *socket* aberto é armazenado num dicionário do servidor chamado *dicionário de sockets abertos*. Esse dicionário relaciona cada ID de cliente ao *socket* que representa a conexão atualmente aberta com o cliente aguardando transmissão de resposta.

Nesse momento, o ator a inicializa e decide enviar uma mensagem m ao ator de recepção. Para isso, uma vez que a mensagem é remota, deposita-a na fila de mensagens Q e notifica a variável de condição associada à fila. O envio é assíncrono, então o ator a não permanece bloqueado, continuando sua execução normalmente.

A notificação da variável de condição associada à fila Q fará com que a thread T_S, que está bloqueada, entre em execução. Esta thread retira a mensagem m da fila (6) e imediatamente abre uma conexão com o servidor, efetuando uma requisição de envio (7).

O servidor, ao receber a requisição de envio, observa que o destinatário de m reside no próprio servidor e efetua a entrega local (8). Convém lembrar que, se o destinatário fosse remoto, o comportamento do servidor seria fazer o *forwarding*, que seria obtido simplesmente por meio da inserção da mensagem m na fila Q do servidor, como explicaremos na próxima seção.

O ator b recebe a mensagem m, efetua algum processamento relevante, e decide responder ao ator remetente com a mensagem r. A partir deste ponto, ocorre o processo inverso: b deposita a mensagem r na fila Q do servidor (9), sinalizando a variável de condição associada. A thread T_S do servidor é despertada e retira a mensagem r da fila (10).

Neste ponto, o comportamento de T_S no servidor é ligeiramente diferente do comportamento no cliente. Ao invés de abrir uma conexão com o destinatário (que não é possível no protocolo HTTP), a thread procura o *socket* apropriado no *dicionário de sockets abertos* do servidor, e então envia a esse *socket* uma resposta HTTP com r no corpo da resposta.

De volta ao lado do cliente, ao receber a resposta HTTP, a thread T_R é despertada, recupera a mensagem r a partir dos dados binários recebidos, e entrega-a ao ator a. Observe que nesta figura, para propósitos de simplicidadade, estamos omitindo o processamento interno de envio e recepção de mensagens efetuado dentro do próprio ator, que já foi ilustrado na Figura 3.1.

Também adotamos, por simplificação, ocultar as *threads de trabalho* do servidor no diagrama. Como explicamos anteriormente, essas threads são as responsáveis por lidar com cada requisição recebida por T_L para que T_L fique disponível o mais breve possível para lidar com novas requisições. Portanto, a maioria do trabalho que o diagrama atribui a T_L é, em realidade, efetuado por *threads de trabalho*.

3.14 Percurso de uma Mensagem com *Forwarding*

Como já mencionamos, o algoritmo que implementamos permite que o servidor efetue *forwarding*, isto é, o encaminhamento de mensagens enviadas por um cliente a outro cliente. A Figura 3.3 ilustra o comportamento dos diversos componentes dos clientes e do servidor no caso de um ator a residente num cliente (Cliente 1) enviar uma mensagem m endereçada a um ator c residente em outro cliente (Cliente 2).

O cliente 1 envia a mensagem m ao servidor normalmente. Mas o servidor, ao invés de efetuar a entrega local, insere a mensagem m na fila de envios remotos Q diretamente (1), a partir de onde a thread T_S a lerá (2), entregando-a na resposta HTTP ao cliente 2 (3), que, finalmente, efetuará a entrega local da mensagem ao ator destinatário c.

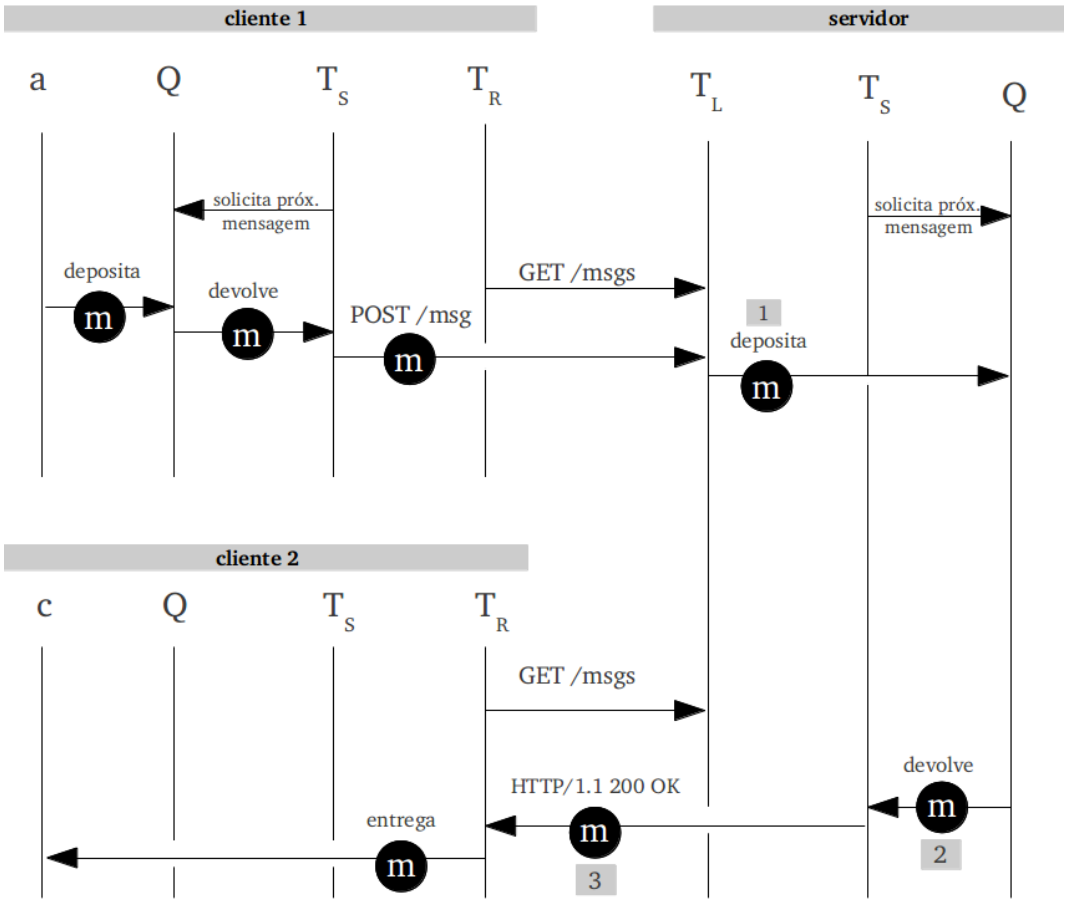

Figura 3.3: *Percurso de uma mensagem com* forwarding

3.15 Protocolo de Comunicação

Como explicado anteriormente, o protocolo de comunicação utilizado no PAWEB é construído sobre o protocolo HTTP [FGM+99], pois desejamos aproveitar a infraestrutura existente compatível com esse protocolo, como por exemplo a possibilidade de utilizar servidores *proxy*, entre outros. Além disso, o uso do HTTP torna possível utilizar o PAWEB diretamente num navegador, tornando a experiência mais fácil para o usuário. Naturalmente, a necessidade da instalação de um complemento de navegador para exibir o conteúdo do PAWEB prejudica esse objetivo, mas qualquer outra alternativa nos pareceu inviável: sem um complemento específico para o PAWEB, a execução do código do cliente está restrita ao que o HTML e a linguagem Javascript podem expressar. Isto conflita com um dos propósitos do PAWEB, que é evitar que o programador tenha que lidar com a situação na qual a linguagem de programação usada para o cliente é diferente da linguagem de programação utilizada para o servidor, situação essa que resulta na necessidade de conversões para transmitir objetos e outros dados.

O protocolo de trasmissão de mensagens não necessita levar em consideração a estrutura interna da mensagem em si. Para fins do protocolo, a mensagem é uma sequência de *bytes* arbitrária que precisa ser transmitida para o sistema remoto. Além do conteúdo da mensagem, é necessário especificar o endereço do ator destinatário.

Portanto, quando um ator no cliente envia uma mensagem para um ator no servidor, a plataforma criará um envelope que conterá o endereço do ator remetente e o do destinatário, bem como o conteúdo da mensagem em si. Conforme sugerido pelo padrão HTTP [FGM+99], o método "POST" é o mais apropriado para esse tipo de comunicação. Portanto a estrutura da mensagem será:

```
POST / HTTP/1.1
Host: <endereço-do-servidor>
Date: <data>
Connection: close
Content-Type: application/octet-stream
Content-Length: <comprimento>

<endereço do ator remetente>
<endereço do ator destinatário>
```

```
<dados>
```

Observe que utilizamos o *MIME type* genérico para dados binários, que é o `application/octet-stream`, pois são dados com estrutura interna opaca ao protocolo. Para a comunicação em sentido contrário, isto é, quando o servidor deseja enviar uma mensagem ao cliente, o que ocorrerá é que o servidor utilizará a "requisição pendente" que o cliente sempre deixa ativa no servidor. Portanto, utilizará o formato de resposta HTTP [FGM+99] para codificar sua mensagem. Desta forma, a estrutura da mensagem será como ilustrado abaixo:

```
HTTP/1.1 200 OK
Date: <data>
Content-Type: application/octet-stream
Content-Length: <comprimento>

<endereço do ator remetente>
<endereço do ator destinatário>
<dados>
```

No lado remoto, uma vez lida e decodificada, a mensagem é encaminhada para o ator destinatário normalmente por meio da fila de mensagens, isto é, o programador que escreve o código do ator não precisa tratá-la de maneira especial.

3.16 Transmissão de Funções

Um dos problemas que precisamos considerar é que os dados transmitidos entre atores, sejam eles locais ou remotos, podem incluir funções. Afinal, numa linguagem funcional (ou que, como no caso do Python, permite a programação funcional embora não seja esse o enfoque da linguagem), as funções são valores de primeira-ordem, o que significa que podem ser tratados como qualquer outro tipo de dados.

Esse fato é relevante devido à pressuposição inicial que adotamos, de que qualquer mensagem trocada entre dois atores pode ser *seriada*, isto é, convertida para uma sequência de *bytes* e depois recuperada dessa sequência. Isto é necessário especialmente para a transmissão de mensagens entre o servidor e o cliente.

Por esse motivo, é necessário que as funções sejam *seriáveis*. A maneira usual de realizar a seriação em Python é utilizar o módulo `pickle` [Pytb].

A princípio poderíamos esperar que o resultado de aplicar esse módulo numa função fosse uma representação binária do código compilado da função, de forma que essa representação pudesse ser utilizada, numa outra máquina, para reconstituí-la e executá-la. Porém, infelizmente não é isso que ocorre. Ao invés disso, a representação da função é simplesmente uma referência a seu nome [Pyta]. Uma consequência lógica desse comportamento é que funções anônimas tipo `lambda` não podem ser seriadas, e nem as funções dinâmicas obtidas, por exemplo, como valor devolvido de uma outra função. Assim, o único tipo de função que podemos seriar são as funções estáticas.

Porém, observamos também que não há razão para seriar uma mensagem que será trocada entre atores residentes na mesma máquina, seja no servidor ou no cliente, pois a mensagem pode ser passada diretamente como objeto Python. Assim, nesse caso específico, referências a funções anônimas e dinâmicas seriam corretamente transmitidas. A única situação problemática ocorre quando um ator numa máquina (servidor ou cliente) necessita enviar uma mensagem que contém uma função dinâmica a um ator residente no lado remoto.

Tínhamos portanto uma decisão importante a tomar no projeto da plataforma:

1. Permitir a passagem de funções anônimas e dinâmicas em mensagens entre atores na mesma máquina, mas proibí-las em mensagens entre atores em máquinas diferentes; ou

2. Proibir completamente a passagem de funções anônimas e dinâmicas em mensagens, já que a princípio um ator não precisa (e talvez nem possa) saber se o ator destinatário de uma mensagem é local ou remoto.

Embora a opção (1) forneça uma flexibilidade muito grande e difícil de descartar, a opção (2) é claramente mais consistente com os princípios gerais da plataforma, que enfatizam a transparência para o programador. Portanto, foi essa a abordagem que adotamos.

É oportuno observar também que o fato do código da função não estar incluído na mensagem não causa problemas para a troca de mensagens entre o servidor e o cliente, já que o nome da função é suficiente para o outro lado executá-la. Isso se deve ao carregamento inicial do código feito na inicialização do cliente, que faz com que tanto o servidor quanto o cliente possuam o mesmo espaço de nomes de módulos e funções.

3.17 Assuntos Não Contemplados

O escopo desse trabalho é apresentar uma implementação inicial do PAWEB como uma "prova de conceito." O objetivo não é, portanto, desenvolver um produto completo pronto para uso mas sim um protótipo funcional que demonstre as características do sistema, porém com algumas limitações, que explicaremos a seguir.

3.17.1 Segurança do Código

Uma vez que o objetivo futuro do PAWEB é funcionar como forma de executar aplicativos remotos através da rede, a preocupação com a segurança é indispensável a uma versão final do projeto. Porém, como no protótipo preferimos concentrar os esforços na implementação dos conceitos fundamentais do sistema, decidimos deixar o desenvolvimento da política de segurança para uma fase futura.

Um dos problemas de segurança mais graves do PAWEB, em sua implementação inicial, é que o servidor pode, a princípio, enviar qualquer código em Python para o cliente executar. Esse código arbitrário pode fazer quaisquer chamadas ao sistema e portanto pode manipular e alterar arquivos, ler informações da conta do usuário, entre outras ações potencialmente maliciosas. Por isso, sem que haja um modelo de segurança cuidadosamente planejado, a utilização num ambiente em que não há confiança mútua (Internet, por exemplo) é atualmente inviável.

3.17.2 Segurança do Protocolo

Um outro problema relacionado à segurança é que o protocolo do PAWEB não usa criptografia, isto é, os dados trafegam pela rede de forma acessível a qualquer leitor que intercepte a conexão entre o cliente e o servidor. Naturalmente, isto não é problemático se o aplicativo sendo construído não lida com dados sensíveis ou confidenciais. Porém, nos dias de hoje a maioria das aplicações *web* necessitam de autenticação de usuário, e nesse caso nossa implementação inicial do PAWEB enviaria o nome de usuário e a senha ao servidor sem criptografia, o que é um enorme risco à segurança.

3.17.3 Interface com o Usuário

O protótipo não implementa uma interface com o usuário além da leitura e exibição de texto, que é suficiente para propósitos de demonstração. Porém, não é suficiente para implementar um aplicativo de produção, meta que está fora do escopo do presente trabalho.

3.17.4 Interface com o DOM

Como explicado anteriormente, o PAWEB, na sua versão inicial, será executado no modo "página inteira," isto é, sem um documento em HTML que o circunda. Porém, poderia ser desejável que o PAWEB também pudesse ser executado no modo "embutido," isto é, como componente de uma página HTML. Nesse caso, seria conveniente que o PAWEB fornecesse meios para interagir com os elementos do documento, e para isso seria necessária uma interface com o *DOM - Document Object Model*, que, como explica a especificação [W3C98], consiste numa interface que permite que programas e *scripts* interajam com o conteúdo, estilo e estrutura dos documentos.

Capítulo 4

Conclusão

Apresentamos neste trabalho uma visão geral do projeto PAWEB, uma plataforma para programação e execução de aplicativos *web* utilizando o modelo de atores. Fornecemos uma breve visão histórica do modelo de atores com uma sucinta explicação de seu funcionamento, mencionando alguns dos vários trabalhos que procuraram implementar características do modelo de atores em diversas linguagens e sistemas. Apesar da maioria das linguagens e plataformas utilizadas na indústria não fornecer suporte direto ao modelo de atores, explicamos a conveniência de integrar esse modelo ao desenvolvimento de aplicativos *web*, conveniência essa que motivou a criação do PAWEB.

Explicamos a arquitetura do sistema e mencionamos as numerosas dificuldades técnicas que enfrentaremos na implementação, oriundas das restrições impostas pela linguagem, pela plataforma e pelo protocolo de comunicação. Porém, como explicamos, cada um desses obstáculos pode ser superado com o uso de certas técnicas e combinações de técnicas já presentes nas aplicações *web* atuais e em várias linguagens de programação, entre as quais nossa principal fonte de inspiração foi a linguagem Erlang. Também fornecemos uma descrição detalhada da implementação e exemplificamos como os vários componentes interagem entre si para permitir a criação de atores, o envio e recebimento de mensagens, a transparência entre entregas de mensagens locais e remotas, e todas as outras características da plataforma.

4.1 Comparação com Trabalhos Relacionados

Em primeiro lugar, o PAWEB não é uma nova linguagem e sim uma plataforma desenvolvida dentro de uma linguagem existente (Stackless Python [Chra]). Esta característica o separa de trabalhos como o Erlang [Eri], o Stage [AE09]

e do SALSA [VA01], que são linguagens em si, embora os dois últimos sejam pré-processadores que convertem o programa para linguagens anteriormente existentes (respectivamente Python e Java). Porém, uma vez que a funcionalidade fornecida é mais importante que a forma sintática na qual ela se reveste, não faremos, na explicação a seguir, uma distinção rígida entre linguagens e plataformas.

O Erlang, como explica Armstrong [Arm07], permite que os atores residam em vários servidores diferentes numa rede e se comuniquem por mensagens de maneira praticamente transparente ao programador. O PAWEB fornece uma funcionalidade similar, que é a transmissão transparente de mensagens entre o cliente e o servidor sem onerar o programador com a implementação dos detalhes da comunicação. Porém, de maneira diferente do Erlang, o PAWEB exige uma topologia mais rígida, pois não permite a construção de um aplicativo distribuído que utiliza atores em múltiplos servidores. Ao invés disso, todos os atores do lado do servidor residem numa mesma máquina virtual, assim como os atores de cada cliente, embora não haja limite teórico ao número de clientes diferentes interagindo com o mesmo servidor. Por outro lado, como os atores Erlang só podem ser executados em máquinas virtuais Erlang, e como não existe plugin Erlang para navegadores, é impossível desenvolver uma aplicação *web* na qual atores são executados do lado do cliente.

A linguagem Stage, como explicam Ayres e Eisenbach [AE09], implementa um modelo de atores sobre a máquina virtual Java, enfatizando a reconfiguração dinâmica sobre uma rede. Uma das diferenças importantes entre o PAWEB e o Stage é que neste os atores são implementados como classes Python nas quais cada método é o "nome" de uma mensagem que o ator tem a capacidade de interpretar [AE09]. Por outro lado, o PAWEB utiliza uma construção sintática de recepção de mensagens inspirada no Erlang, linguagem na qual o ator especifica padrões e comportamentos associados à recepção de mensagens que casam com cada padrão. Isto permite que os atores mudem de comportamento (isto é, mudem os tipos de mensagens aceitas) durante sua execução sem a necessidade de mudar de "classe Python" como ocorre no Stage. Uma outra vantagem do casamento de padrões do PAWEB é que esta abordagem não impõe uma forma fixa às mensagens, isto é, não existe um campo "nome" para identificar o método a chamar, como ocorre no Stage. Por outro lado, a desvantagem do PAWEB em relação ao Stage é a mesma que mencionamos no caso do Erlang: não é possível distribuir os atores do servidor em várias máquinas diferentes. Por outro lado, o Stage não permite diretamente a pro-

gramação *web*, pois não prevê a possibilidade de acesso à aplicação a partir de um navegador web.

A plataforma SALSA de Varela et al. [VA01] implementa o envio transparente de mensagens independente de localização, mesmo que os atores migrem diversas vezes ao longo da vida da aplicação. O PAWEB também permite o envio transparente de mensagens, porém não dá suporte à migração de atores nem ao balanceamento de carga. Afinal, o propósito do SALSA é, como os autores descrevem, permitir a criação de aplicações dinamicamente reconfiguráveis, enquanto que o propósito do PAWEB é servir como plataforma para desenvolvimento de aplicativos *web*. O SALSA introduz novas construções de linguagem, particularmente três tipos de continuações: continuações com passagem de token, "join-continuations" e continuações de primeira classe [VA01], enquanto que o PAWEB não introduz nenhuma nova sintaxe, utilizando somente a disponível no Stackless Python. Por último, o código Salsa é compilado na forma de código Java e depois executado na máquina virtual Java, enquanto que o PAWEB executa código nativamente em Python sem a necessidade de conversão de linguagem.

O arcabouço Lift [Pol] propõe a utilização do paradigma de programação funcional para a programação *web*, afinal, como explica Ghosh [GV09], esse paradigma tende a ser mais apropriado para esse ambiente do que a programação imperativa tradicional. O Lift fornece ao programador uma biblioteca para a programação baseada no modelo de atores que traz características mais apropriadas ao ambiente *web* que a biblioteca padrão de atores em Scala [PV10]. O Lift também fornece ao programador uma abstração que oculta do programador os detalhes da conexão HTTP, mas exige a especificação a estrutura do documento HTML [PV10]. Portanto, o fornecimento de uma infraestrutura para o modelo de atores e a ocultação do protocolo HTTP sob uma abstração são características comuns ao Lift e ao PAWEB. Porém, a arquitetura proposta pelo Lift é fundamentalmente diferente da proposta do PAWEB, pois naquela não existem atores residentes no cliente e sim componentes HTML que, por meio de código Javascript (fornecido pela infraestrutura), geram eventos que são interpretados por atores no servidor, como ilustra o exemplo de aplicação apresentado por Pollak et al. [PV10]. Esta abordagem apresenta a distinta vantagem de que o usuário não necessita de um *plugin* de navegador para utilizar a aplicação. Porém, no PAWEB, há a vantagem de que o componente cliente também tem a capacidade de executar atores, isto é, o cliente não é meramente uma interface de apresentação. Esta distinção pode ser importante

porque, no PAWEB, toda a lógica de formatação de informação e organização da interface gráfica, bem como a lógica de interação com o usuário, fica concentrada no processador da própria máquina cliente, deixando no servidor apenas os atores cuja funcionalidade está realmente relacionada com as regras de negócio da aplicação. Dependendo da situação, isso pode evitar uma possível sobrecarga do servidor na presença de um número elevado de clientes. Além disso, em alguns casos o tempo de resposta da interface no PAWEB pode ser menor devido ao fato de que uma quantidade razoável de processamento pode ser realizada localmente sem aguardar uma comunicação do servidor.

O módulo Candygram para a linguagem Python [Hob04] permite o envio e recebimento de mensagens entre *threads* Python utilizando uma semântica quase idêntica à do Erlang [Eri]. Porém, embora auxilie na implementação de aplicações que seguem o modelo de atores, esse módulo só torna possível a comunicação entre *threads* e não estruturas mais leves como as *tasklets* do Stackless Python e também não permite a troca de mensagens entre atores em máquinas diferentes.

Em resumo, verifica-se que o PAWEB apresenta algumas limitações em relação a outros trabalhos, notoriamente a incapacidade de realizar a migração de atores e o balanceamento de carga. No entanto, possui como principais vantagens a facilidade de programação (visto que não se trata de uma linguagem nova e sim de uma utilização do Stackless Python), e a possibilidade da existência de atores no lado do cliente (navegador), e consequentemente o aproveitamento da capacidade de processamento do cliente.

4.2 Trabalhos Futuros

A versão atual do PAWEB, embora suficiente para a demonstração dos conceitos principais, não está apta para utilização num ambiente real. Afinal, faltam-lhe uma série de requisitos de segurança e interface que são essenciais a uma aplicação *web* que é executada num ambiente em que não há plena confiança entre as partes. Desta forma, um dos trabalhos futuros mais importantes seria a implementação de uma política de segurança que permitisse a limitação dos privilégios do código executado no cliente, impedindo-o, por exemplo, de ter acesso ao sistema de arquivos ou a configuração da máquina do usuário livremente. Outra preocupação importante com a segurança é o fato de que os dados que viajam pela plataforma podem conter informações sensíveis, portanto um trabalho futuro possível seria a implementação de um mecanismo de

criptografia.

Outro trabalho futuro importante é o desenvolvimento de uma interface com o usuário para a plataforma, uma vez que na sua versão atual a interação é baseada em texto puro e serve somente para propósitos de demonstração. Além disso, o desenvolvimento de *plugins* do PAWEB para outros navegadores é um bom candidato para um trabalho futuro, já que, na versão atual, o PAWEB só poderá ser executado em navegadores da "família Netscape."

Referências Bibliográficas

[AB88] WC Athas e NJ Boden. Cantor: an actor programming system for scientific computing. Em *Proceedings of the 1988 ACM SIGPLAN workshop on Object-based concurrent programming*, páginas 66–68. ACM, 1988. 9

[Ado] Adobe Inc. Flash CS5. http://www.adobe.com/flash. Acesso em 2010-03-15. 14

[AE09] J. Ayres e S. Eisenbach. Stage: Python with Actors. Em *Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering*, páginas 25–32. IEEE Computer Society, 2009. 6, 11, 15, 18, 57, 59, 85, 86

[Agh86] G.A. Agha. Actors: a model of concurrent computation in distributed systems. *AITR-844*, 1986. 5, 6, 7, 8, 9, 10, 22, 34

[Arm03] J. Armstrong. *Making reliable distributed systems in the presence of software errors*. Citeseer, 2003. 8, 10, 17

[Arm07] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007. 7, 10, 25, 26, 29, 31, 32, 46, 47, 48, 49, 54, 58, 86

[Arn86] J.Q. Arnold. Shared Libraries on UNIX System V. Em *Proceedings of the 1986 Summer USENIX Conference, Atlanta, Ga.*, páginas 395–404, 1986. 13

[AT97] American Telephone and Telegraph, Inc. e The Santa Cruz Operation, Inc. *System V Application Binary Interface, ed. 4.1*. AT&T, 1997. 13

[BDMN73] G. Birtwistle, Ole Johan Dahl, B. Myhrtag e Kristen Nygaard. *Simula Begin*. Auerbach Press, Philadelphia, 1973. 9

[BK93] R. Brown e J. Kyle. *PC interrupts: a programmer's reference to BIOS, DOS and third-party calls*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1993. 3

[Chra] Christian Tismer. Stackless Python. http://www.stackless.com. Acesso em 2010-05-30. 15, 47, 85

[Chrb] Christian Tismer. Stackless Python - Channels. http://www.stackless.com/wiki/Channels. Acesso em 2010-06-17. 49

[Cli81] W.D. Clinger. *Foundations of actor semantics*. Massachusetts Institute of Technology Cambridge, MA, USA, 1981. 4

[CM08] D. Crane e P. McCarthy. *Comet and Reverse Ajax: The Next Generation Ajax 2.0*. Springer, 2008. 45

[Dig83] Digital Equipment Corporation. *EDT Editor Manual*. Digital Equipment Corporation, Maynard, Massachusetts, 1983. 12

[Eri] Ericsson Computer Science Laboratory. Open Source Erlang. http://www.erlang.org. Acesso em 2010-06-10. 1, 5, 6, 7, 10, 11, 15, 18, 23, 46, 50, 52, 58, 85, 88

[FB96] N. Freed e N. Borenstein. RFC 2045: Multipurpose internet mail extensions (MIME) part one: Format of internet message bodies. http://tools.ietf.org/html/rfc2045, 1996. Acesso em 2010-05-27. 14, 21

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach e T. Berners-Lee. Hypertext Transfer Protocol - HTTP/1.1, 1999. 2, 21, 41, 42, 44, 45, 68, 69, 72, 79, 80

[FSF] Free Software Foundation - FSF. Gnu c compiler - gcc. http://gcc.gnu.org. Acesso em 2010-03-29. 13

[Goo] Google Inc. Google chrome web browser. http://www.google.com/chrome. Acesso em 2010-04-01. 75

[GR83] A. Goldberg e D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1983. 9

[Gui03] Guido van Rossum. *The Making of Python*. Artima Developer Online, 2003. http://www.artima.com/intv/pythonP.html. Acesso em 2010-06-02. 15

[GV09] D. Ghosh e S. Vinoski. Scala and Lift—Functional Recipes for the Web. *IEEE Internet Computing*, 13(3):88–92, 2009. 17, 87

[HB77] C. Hewitt e H. Baker. Actors and continuous functionals. *AIM-436a*, 1977. 5

[HBS73] C. Hewitt, P. Bishop e R. Steiger. A universal modular actor formalism for artificial intelligence. Em *Proceedings of the 3rd international joint conference on Artificial intelligence*, páginas 235–245. Morgan Kaufmann Publishers Inc., 1973. 1, 4, 5, 6, 22

[HH10] Ian Hickson e David Hyatt. HTML5: A vocabulary and associated APIs for HTML and XHTML. http://www.w3.org/TR/html5, 2010. Acesso em 2010-06-21. 14

[Hic09] Ian Hickson. The web sockets api w3c working draft. http://www.w3.org/TR/websockets, 2009. Acesso em 2010-06-18. 74

[Hic10] Ian Hickson. The websocket protocol (ietf draft). http://tools.ietf.org/html/draft-ietf-hybi-thewebsocketprotocol-00, 2010. Acesso em 2010-06-18. 74

[HO07] P. Haller e M. Odersky. Actors that unify threads and events. Em *Coordination Models and Languages*, páginas 171–190. Springer, 2007. 5

[Hoa74] C.A.R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974. 4, 9

[Hob04] Michael Hobbs. Candygram. http://candygram.sourceforge.net, 2004. Acesso em 2010-05-15. 18, 88

[Hof99] Larry Hoff. Netscape Plug-Ins. *Linux Journal*, 1999(65es):5, 1999. 12

[Hol00] M.E. Holzschlag. *Special Edition Using HTML 4 - 6th. ed.* Macmillan Press Ltd., Indianapolis, IN, USA, 2000. 14

[IN] Internet Corporation of Assigned Names ICANN e Numbers. Internet Assigned Numbers Authority - IANA. http://www.iana.org. Acesso em 2010-05-30. 14, 21

[ISO] Internet Society ISOC. Internet engineering task force - ietf. http://www.ietf.org. Acesso em 2010-05-11. 42, 74

[KM09] J. Kuuskeri e T. Mikkonen. Partitioning web applications between the server and the client. Em *Proceedings of the 2009 ACM symposium on Applied Computing*, páginas 647–652. ACM, 2009. 74

[Kri01] D.M. Kristol. HTTP Cookies: Standards, privacy, and politics. *ACM Transactions on Internet Technology (TOIT)*, 1(2):198, 2001. 42

[Kri07] S. Krishnamurthi. *Programming languages: Application and interpretation*. Shriram Krishnamurthi (Brown University, USA), 2007. Disponível gratuitamente em: http://www.cs.brown.edu/ sk/Publications/Books/ProgLangs/. 16, 42, 44

[Lie81] H. Lieberman. A preview of act 1. *AIM-625*, 1981. 9

[Lis75] B. Liskov. An introduction to CLU. *New directions in algorithmic languages*, páginas 139–156, 1975. 9

[Mic98] Microsoft Corporation. Active server pages .net - asp.net. http://msdn.microsoft.com/en-us/asp.net, 1998. Acesso em 2010-06-19. 43

[Moza] Mozilla Foundation. Gecko plugin api reference (npapi). https://developer.mozilla.org/en/Gecko_Plugin_API_Reference. Acesso em 2010-05-01. 21, 22

[Mozb] Mozilla Foundation. Mozilla Firefox Web Browser. http://www.mozilla.org/products/firefox. Acesso em 2010-04-12. 20

[Mozc] Mozilla Foundation. Mozilla plugin documentation. https://developer.mozilla.org/en/plugins. Acesso em 2010-06-01. 12, 42

[Net] Netscape AOL Inc. Netscape browser. http://netscape.aol.com. Acesso em 2009-12-15. 12, 13, 20, 42

[O+03] Martin Odersky et al. Scala programming language. http://www.scala-lang.org, 2003. Acesso em 2010-06-16. 11, 17

[OAC+04] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman e M. Zenger. An introduction to Scala. *Programming Methods Laboratory, EPFL, Switzerland*, 2004. 11, 17

[OSV08] M. Odersky, L. Spoon e B. Venners. *Programming in Scala: a comprehensive step-by-step guide*. Artima Inc., USA, 2008. 11

[PG] The PHP Group. Php - hypertext preprocessor. http://www.php.net. Acesso em 2010-03-27. 43

[Pol] D. Pollak. Lift - the simply functional web framework. http://liftweb.net/. Acesso em 2010-06-12. 45, 87

[PV10] D. Pollak e S. Vinoski. A Chat Application in Lift. *IEEE Internet Computing*, 14(3):88–91, 2010. 17, 18, 45, 87

[Pyta] The Python Foundation. Python language reference. http://docs.python.org/lang. Acesso em 2010-06-04. 54, 81

[Pytb] The Python Foundation. Python library reference. http://docs.python.org/library. Acesso em 2010-06-04. 15, 80

[Pyt89] The Python Foundation. Python programming language. http://www.python.org, 1989. Acesso em 2010-06-16. 14

[Rey93] J.C. Reynolds. The Discoveries of Continuations. *Lisp and Symbolic Computation*, 6(3):233–247, 1993. 16

[RLHJ98] D. Raggett, A. Le Hors e I. Jacobs. HTML 4.0 Specification. *W3C REC REC-html40-19980424*, 1998. 14

[Rus06] A. Russell. Comet: Low latency data for browsers. O'Reilly Emerging Technology Conference presentation, March, 2006. 44, 45

[SJS76] G.L. Steele Jr e G.J. Sussman. Lambda: The ultimate imperative. *AIM-353*, 1976. 54

[Ste93] W. Richard Stevens. *Advanced Programming in the UNIX (R) Environment*. Addison-Wesley Professional, 1993. 13, 20

[SUN] SUN Microsystems Inc. Java and solaris threading. http://java.sun.com/docs/hotspot/threads/threads.html. Acesso em 2010-06-18. 46

[Tis00] C. Tismer. Continuations and stackless python. Em *Proceedings of the 8th International Python Conference*. Citeseer, 2000. 15, 16, 17, 28, 48

[TW97] A.S. Tanenbaum e A.S. Woodhull. *Operating Systems: Design and Implementation, 2nd. ed.* Prentice Hall, New Jersey, USA, 1997. 3, 6, 15, 46

[VA01] C. Varela e G. Agha. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices*, 36(12):34, 2001. 6, 11, 18, 86, 87

[vW66] A. van Wijngaarden. Recursive definition of syntax and semantics. Em *Formal Language Description Languages for Computer Programming: Proceedings of the IFIP Working Conference on Formal Language Description Languages*, páginas 13–24, 1966. 16

[W3C98] World Wide Web Consortium W3C. Document object model (dom) level 1 specification. Recommendation REC-DOM-Level-1-19981001, 1 Oct 1998, 1998. 83

[YNL09] Y. Yu, L. Ning e W. Liu. Combining HTML 5 with MVC Framework to Simplify Real-Time Collaboration for Web Development. Em *2009 International Conference on Multimedia Information Networking and Security*, páginas 29–32. IEEE, 2009. 74