PUC-Rio - Certificação Digital Nº 0321108/CA

3
Modelo Proposto

O objetivo principal deste trabalho é disponibilizar para os desenvolvedores de aplicações ferramentas que permitam a construção de sistemas orientados a eventos de forma portável, flexível e com desempenho adequado às finalidades de seus projetos. Para isso, procuramos definir um modelo que sirva como base para o desenvolvimento de um escalonador de eventos que encapsule a demultiplexação e despacho dos eventos para seus respectivos tratadores.

O primeiro modelo estudado durante o desenvolvimento desse trabalho baseia-se no padrão reativo apresentado no *Reactor*[28], onde o despacho dos eventos para seus tratadores é executado através de métodos síncronos, fazendo com que a chamada não retorne até o término do tratamento do evento. Embora de simples compreensão e implementação, quando são executados fora de um ambiente multiprocessado, os modelos síncronos se mostram ineficientes em serviços de grande escala, devido ao uso não otimizado do tempo disponível de processamento, pois o fluxo de processamento seqüencial em uma única linha de execução do tratador do evento impede que o controle seja devolvido ao componente *Reactor*, impossibilitando o recebimento de novos eventos, degradando o desempenho do sistema.

A utilização de um ambiente com múltiplas linhas de execução permite minimizar os efeitos do despacho síncrono dos eventos no modelo reativo, porém, nesse caso, a programação perde o seu fluxo linear e passa a depender de um maior controle na sincronização e processamento dos dados e requisições, fazendo com que o desenvolvimento de serviços se torne menos trivial, até mesmo para desenvolvedores experientes. Este modelo é observado no *Proactor*[28], onde o processamento de operações de maneira assíncrona simula um ambiente com múltiplas linhas de execução.

Nossa proposta é aliar a flexibilidade oferecida pelo padrão *Reactor* para o tratamento de múltiplas fontes de eventos com o desempenho conseguido através do processamento assíncrono dos tratadores de eventos observado na implementação do padrão *Proactor*. Para evitar a complexidade

do gerenciamento do fluxo de dados e manutenção do contexto de execução, os eventos recebidos serão executados em corrotinas *stackfull*, garantindo assim a possibilidade de interrupção e retomada do processamento com a troca de contexto sendo encapsulada dentro do mecanismo de corrotinas.

O modelo da solução proposta é composto por três componentes principais. O *Observador de Eventos* encapsula um demultiplexador de eventos responsável por armazenar e aguardar mudanças no estado de um conjunto de fontes de eventos. Ele interage diretamente com o *Repositório de Tratadores de Eventos* (*Routine Pool*), componente responsável pelo armazenamento dos tratadores de eventos e por associar cada fonte ao seu tratador específico. O *Escalonador* fornece a API disponível para o desenvolvedor de aplicações e é responsável por gerenciar a complexidade do escalonamento dos métodos assíncronos, alimentando o *Repositório de Tratadores de Eventos* com novos tratadores e despachando os métodos prontos para o processamento.

Nos diagramas apresentados nas figuras 3.1 e 3.2 observa-se a dinâmica de colaboração dos três componentes do modelo, que se inicia com o registro das fontes de evento e de seus respectivos tratadores no escalonador, que por sua vez armazena o tratador no *Repositório de Tratadores de Eventos* e registra cada fonte de eventos no *Observador de Eventos*. Em seguida, inicia-se o ciclo de processamento da aplicação com eventuais sinalizações de eventos coletadas pelo *Observador de Eventos*, fazendo com que o *Repositório de Tratadores de Eventos* insira o tratador correspondente em uma fila de prontos. Os eventos são consumidos mediante requisições explícitas feitas pelo *Escalonador*, que recebe os tratadores em estado pronto para execução e (re)inicia seu processamento. Neste capítulo é feito um detalhamento mais profundo de cada um desses componentes, identificando sua estrutura e seus métodos de comunicação.

3.1 Observador de Eventos

O módulo *Observador de Eventos* atua como a interface entre os componentes do escalonador e as fontes de eventos. Cabe a ele encapsular a complexidade de demultiplexação de eventos provenientes de diferentes fontes e associá-los aos identificadores dos objetos que encapsulam os tratadores concretos de eventos. Para tratar a heterogeneidade das fontes de eventos, esse componente define uma abstração *Event Source*, que permite integrar as fontes de origens assíncronas, fontes de eventos temporizadas e

PUC-Rio - Certificação Digital Nº 0321108/CA

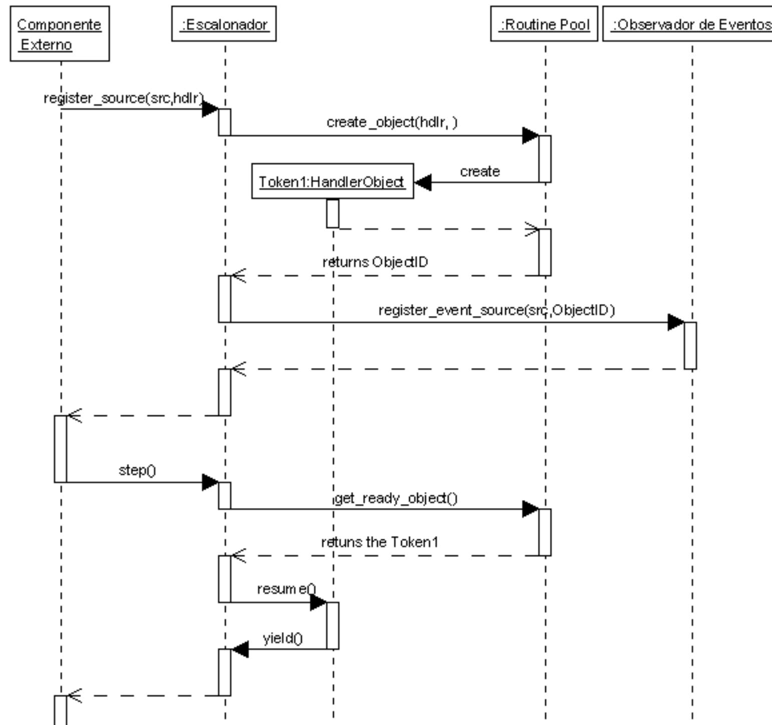

Figura 3.1: Diagrama de seqüência do Escalonador.

PUC-Rio - Certificação Digital Nº 0321108/CA

também possibilita a integração de fontes de origem síncrona executando sobre um ambiente *multithread*.

O *Event Source* funciona de acordo com o padrão de projeto *Observador*[12], onde as fontes devem prover métodos para o registro do observador, que deve ser notificado de mudanças no estado da fonte de eventos, evitando a necessidade de uma espera ocupada (*busy wait*). É comum, em APIs assíncronas, a existência de métodos para o aguardo de notificações, tornando imediata sua integração ao *Observador de Eventos*. Os temporizadores são eventos cuja fonte enviará uma notificação ao observador sempre que o sistema detectar expiração do tempo determinado pelo usuário. É importante lembrar que em um sistema colaborativo, depende-se da liberação espontânea dos recursos alocados a cada rotina, o que faz com que não exista garantia de que os temporizadores definidos serão executados no exato momento de sua expiração. Essa condição é difícil de ser garantida até mesmo em sistemas preemptivos.

A estrutura de componentes do *Observador de Eventos* segue o modelo proposto pelo padrão *Reactor*, tendo sua estrutura de execução baseada em um demultiplexador de eventos, responsável por aguardar notificações de eventos provenientes das diferentes fontes registradas pelo usuário. É comum em alguns casos, como por exemplo a API de *sockets*, a existência de

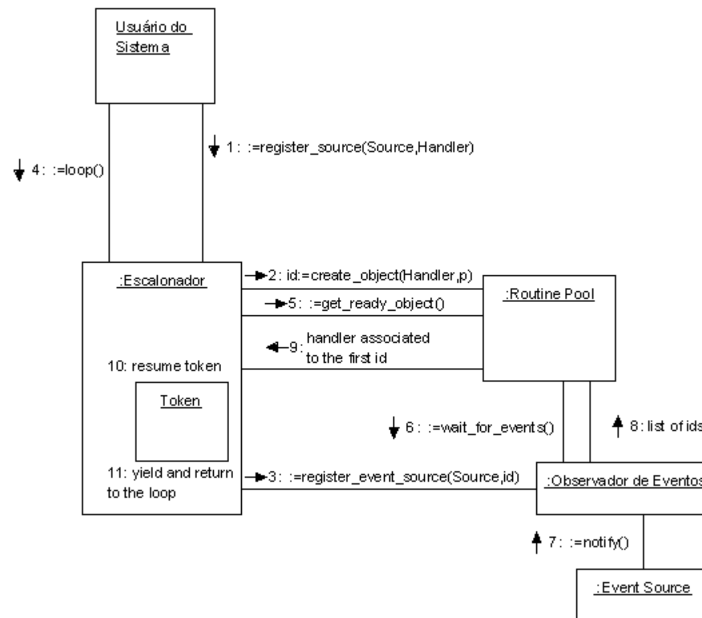

Figura 3.2: Diagrama de colaboração do modelo apresentado.

PUC-Rio - Certificação Digital Nº 0321108/CA

um demultiplexador de eventos síncrono, que recebe um conjunto de fontes de evento (nesse caso *sockets*) e aguarda bloqueado até que algum evento seja recebido em uma das fontes registradas ou que se esgote um tempo limite definido. Uma operação bloqueante para a verificação de mudança de estado nas fontes de eventos pode impedir o recebimento de notificações provenientes de outras fontes, colocando em risco o desempenho do sistema. Por isso é fundamental que esse demultiplexador ofereça meios de definir limites de tempo para essa operação bloqueante, minimizando assim a degradação do sistema devido a um bloqueio por intervalos de tempo longos.

No diagrama de classes do *Observador de Eventos* ilustrado na figura 3.3, foram utilizadas, a título de ilustração, duas derivações da interface *Event Source*, uma implementando uma fonte de eventos provenientes da comunicação de rede, utilizando *sockets*, e outra implementando temporizadores.

O fluxo de dados do *Observador de Eventos* é apresentado na figura 3.4, onde pode-se observar a atuação inicial de componentes externos na configuração do observador, registrando uma ou mais fontes de eventos. Ao ser estimulado através do método `wait_for_events()`, o observador inicia a tarefa de demultiplexação dos eventos recebidos, retornando uma lista de identificadores de eventos prontos para serem tratados.

O demultiplexador de eventos interage diretamente com o sistema operacional, e o seu funcionamento depende das primitivas oferecidas pelo sistema operacional para a manipulação de cada fonte de eventos, que

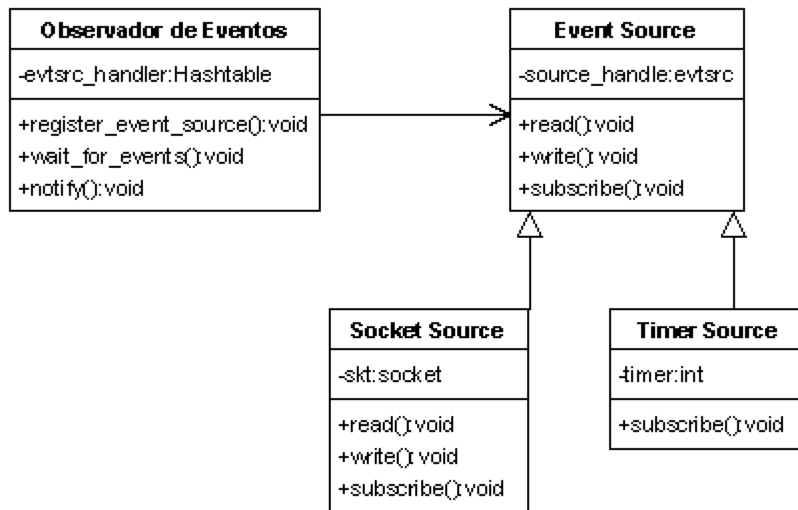

Figura 3.3: Diagrama de classes do Observador de Eventos.

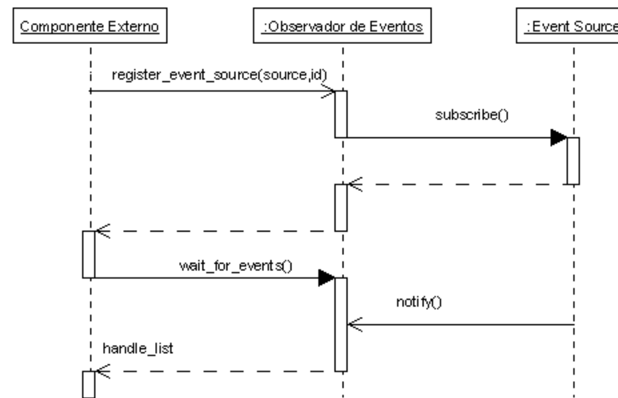

Figura 3.4: Diagrama de seqüência do Observador de Eventos.

PUC-Rio - Certificação Digital Nº 0321108/CA

podem ser divididas em três grupos:

- **Fontes Assíncronas** - As fontes assíncronas são aquelas que permitem o registro de *callbacks* para receber as notificações de mudança de estado. São as que melhor se integram ao modelo pois não há necessidade de bloquear a execução de outros processamentos para verificar a mudança no estado das fontes.
- **Fontes Síncronas** - São as fontes onde o método de aguardo de notificações de evento pode bloquear por tempo indeterminado. Estão normalmente presentes nos sistemas pois apresentam um modelo de programação linear, porém sem garantias no desempenho global da aplicação. Este grupo, necessita de estruturas de apoio para se integrar ao modelo proposto. Uma possível solução é encapsular a fonte síncrona dentro de uma *thread* individual, e usar algum

mecanismo de comunicação entre as duas *threads* para a passagem de eventos de mudança de estado.

- **Fontes Síncronas com tempo de expiração** - Assim como as fontes síncronas sem tempo de expiração, deve-se chamar explicitamente o método para verificação de mudança no estado da fonte de eventos, porém com um parâmetro extra, que garante um limite máximo para o tempo de bloqueio do método.

Dentro deste modelo, múltiplos demultiplexadores de eventos podem ser combinados dentro da estrutura do *Observador de Eventos*. Os demultiplexadores de fontes síncronas com tempo de expiração podem ser executados serialmente controlando o tempo que o sistema permanece bloqueado para que este não se torne muito grande e impeça o recebimento de notificações vindas de fontes assíncronas.

PUC-Rio - Certificação Digital Nº 0321108/CA

3.2 Repositório de Tratadores de Eventos

O *Repositório de Tratadores de Eventos* recebe os eventos do *Observador de Eventos* e identifica os objetos tratadores correspondentes a cada fonte de eventos. O gerenciamento dos tratadores nesse componente é feito através de duas coleções: Uma tabela, associando os tratadores aos seus respectivos identificadores de fontes de eventos (*ObjectPool*), e uma *Fila de Prontos*, que armazena os tratadores com eventos disponíveis para processamento. O *Repositório de Tratadores de Eventos* define uma estrutura base de objeto, o *HandlerObject*, que possui uma referência para o método responsável pelo tratamento do evento e um conjunto de parâmetros, acessíveis durante a execução do método.

O *Repositório de Tratadores de Eventos* mantém uma fábrica de objetos *HandlerObject* acessível através do método `create_object()`, responsável por encapsular os parâmetros recebidos e associar um identificador único ao novo objeto criado. Esse objeto é então inserido no conjunto *ObjectPool* associado ao seu identificador. Esse identificador deverá ser registrado no *Observador de Eventos* juntamente com uma fonte de eventos e será utilizado como forma de identificar unicamente o objeto responsável pelo tratamento do evento. O diagrama de classes do *Repositório de Tratadores de Eventos* está ilustrado na figura 3.5.

O modelo produtor/consumidor[8, 11] é utilizado para gerenciar a organização do fluxo de objetos nas duas estruturas de armazenamento do

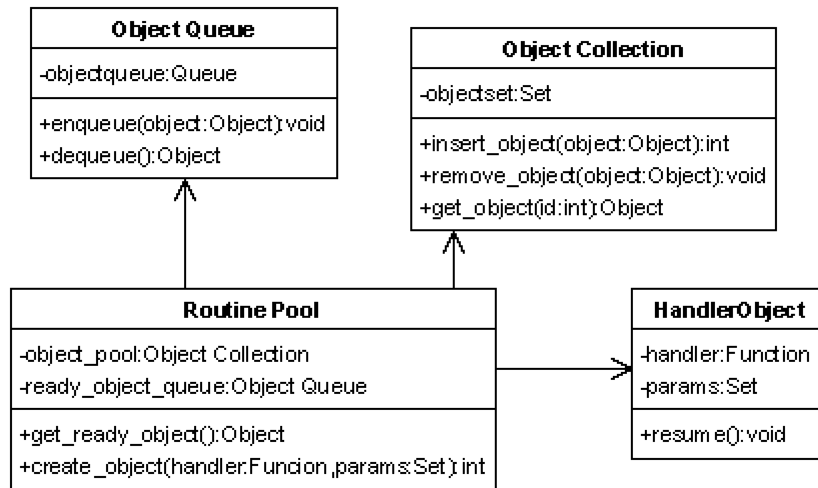

Figura 3.5: Diagrama de classes do Repositório de Tratadores de Eventos.

Repositório de Tratadores de Eventos. O método `get_ready_object()` funciona como consumidor da fila de objetos prontos para processamento, entregando o primeiro elemento da fila. O próprio componente é responsável por alimentar a *Fila de Prontos* quando recebe a requisição de recuperação de objetos e não há nenhum disponível na fila. Nesse momento, o *Repositório de Tratadores de Eventos* se comunica diretamente com o *Observador de Eventos* para obtenção de novos eventos através do método `wait_for_events()`. O fluxo seqüencial de execução é ilustrado na figura 3.6.

PUC-Rio - Certificação Digital Nº 0321108/CA

3.3 Escalonador

O Escalonador é o componente responsável por encapsular a complexidade do gerenciamento do processamento concorrente de tarefas assíncronas, com o intuito de minimizar o tempo em que o escalonador permanece bloqueado aguardando a finalização do tratamento de eventos. O modelo de funcionamento do *Escalonador* é baseado no padrão de projeto *Proactor* aliado ao padrão de projeto *Assincronous Completion Token* ou ACT[28]. O padrão ACT utiliza um *Token de Execução* com informações do contexto de execução de métodos e se encaixa em um cenário onde a seqüência do processamento de métodos que executam operações assíncronas depende do valor de retorno dessas operações.

É de responsabilidade do *Escalonador* gerenciar as trocas de contexto associadas com as sucessivas retomadas de processamento de cada tratador individual. Para isso, utiliza-se um modelo de corrotinas *stackfull* as-

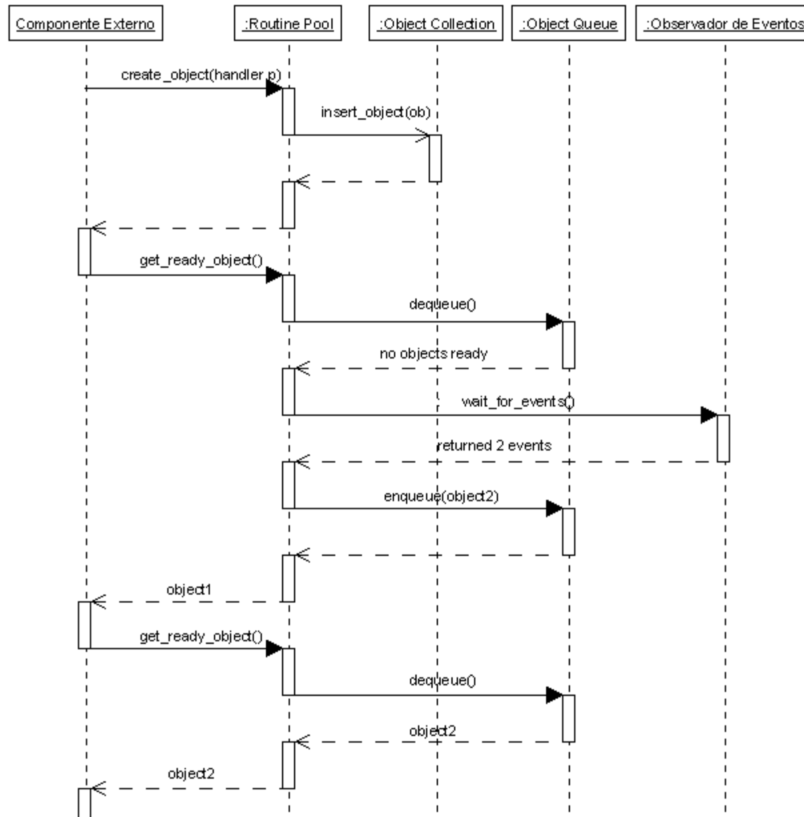

Figura 3.6: Diagrama de seqüência do Repositório de Tratadores de Eventos.

simétricas, que tem como característica principal a possibilidade de suspender sua execução mesmo dentro de métodos aninhados, mantendo a pilha de execução como parte da corrotina, como visto na seção 2.2.3. No *Escalonador*, todo objeto tratador de evento será encapsulado em uma corrotina, representando o *Token de Execução*, passado para o *Repositório de Tratadores de Eventos* como *HandlerObject*.

O *Escalonador* também é responsável por oferecer a interface de programação para a aplicação, composta por um método que permite o registro de um par associativo, com uma fonte de eventos e um método tratador (`register_source()`) e dois métodos que controlam a execução: o `step()`, que faz com que no máximo um evento seja tratado e o `loop()`, que permite a execução contínua do processamento dos eventos, executando até a sinalização de parada por parte da aplicação, através da chamada ao método `stop()`. Na figura 3.7 encontra-se o diagrama de classes do *Escalonador*.

Após ser iniciado, o *Escalonador* recebe as requisições de registro de fontes de eventos associadas a métodos tratadores. Os métodos são encapsulados em *Tokens* e alimentam o *Repositório de Tratadores de Eventos*, que

PUC-Rio - Certificação Digital Nº 0321108/CA

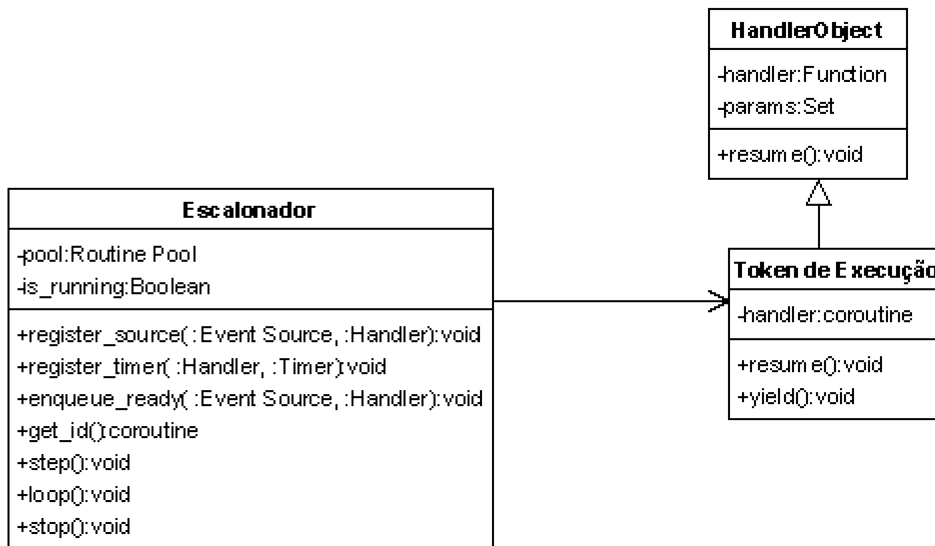

Figura 3.7: Diagrama de classes do Escalonador.

por sua vez devolve identificadores únicos para os objetos criados. Esses identificadores são utilizados para registrar as fontes de evento no *Observador de Eventos*. Após o processo de registro, o escalonador está pronto para entrar em execução, iniciando o ciclo de processamento e escalonamento dos eventos.

A cada passo do ciclo do *Escalonador*, ele interage diretamente com o *Repositório de Tratadores de Eventos* requisitando objetos prontos para o reinício do processamento não bloqueante. É de responsabilidade do *Repositório de Tratadores de Eventos* a comunicação com o *Observador de Eventos* caso não haja objetos prontos para processamento em sua fila de prontos. Ao receber o objeto do repositório, o escalonador retoma o processamento do tratador através do método `coroutine.resume()`, passando os parâmetros previamente registrados no próprio objeto.

O *Escalonador* trabalha em um modelo colaborativo sem suporte a preempção. Logo, uma vez retomado o processamento, é responsabilidade da tarefa executada devolver explicitamente o controle para o escalonador através do método `coroutine.yield()`, ou simplesmente terminar sua execução. Adicionalmente podem ser retornadas sinalizações para o módulo *Escalonador*, como o registro de novas fontes de evento, registro de finalizadores associados ao término da corrotina ou um evento temporizado.

PUC-Rio - Certificação Digital Nº 0321108/CA

3.4
Considerações Finais

O modelo proposto nesse capítulo apresenta um alto grau de generalização nas estruturas definidas, permitindo sua implementação de maneira estruturada em diversas linguagens de programação. Nos capítulos 4 e 5 são apresentados dois casos de uso, com implementações do modelo proposto em ambientes bastante distintos.

O primeiro ambiente, apresenta a integração com o OiL, uma implementação em Lua[27] do padrão CORBA, onde existe a oportunidade de se efetuar chamadas remotas de métodos usando uma API de comunicação de rede por meio de *sockets* originalmente síncronos. O segundo ambiente analisa o modelo de desenvolvimento para dispositivos celulares; um ambiente nativamente assíncrono onde a dificuldade reside em gerenciar o fluxo de informações entre as diversas fontes de eventos e seus respectivos tratadores, oferecendo um estilo de programação síncrono através do escalonamento colaborativo de múltiplas linhas de execução.

A utilização da linguagem Lua permite simplificar a descrição de dados e estruturas de armazenamento, simplificando diversos detalhes de implementação do modelo de escalonamento proposto. Com o uso de tabelas associativas, temos facilmente um mapeamento direto entre fontes de eventos e seus respectivos tratadores no *Repositório de Tratadores de Eventos*. Além disso, a existência de suporte à corrotinas assimétricas, *stackfull* e como objetos de primeira classe na linguagem, simplifica a implementação de um modelo de múltiplas linhas de processamento, permitindo que o contexto de cada tratador de eventos seja mantido pela própria corrotina, sem a necessidade de realizar grandes trocas de contexto no módulo *Escalonador*. Lua também oferece uma API de *sockets*[4, 17], utilizada na implementação do *Observador de Eventos*, com suporte a definição de limites de tempo de bloqueio e com alto grau de portabilidade em sistemas UNIX e Windows.

PUC-Rio - Certificação Digital Nº 0321108/CA