4
Arquitetura de Componentes para o OiL

Como já foi discutido, o desenvolvimento do OiL é interessante para a experimentação de técnicas de construção de *middlewares* e para a criação de protótipos de serviços usando a infra-estrutura básica provida pelo OiL para a realização de chamadas remotas. Além disso, ele foi a base utilizada em (53) para a pesquisa em adaptações no nível da aplicação.

No entanto, a implementação do OiL foi efetuada sem levar em consideração a possibilidade de adaptação dentro do próprio *middleware*, preocupando-se mais com a conformidade com o padrão CORBA. Por isso, a separação de unidades funcionais não era boa o suficiente para que se pudesse simplesmente trocar a implementação de uma de suas partes por outra, para mudar um protocolo ou para selecionar o tipo de codificação. Propomos, neste trabalho, uma separação de interesses mais bem definida, de forma a facilitar adaptações futuras no nível do *middleware*. Procuramos, para isto, experimentar várias das idéias apresentadas no capítulo 2.

Utilizamos para essa separação de interesses o modelo de componentes apresentado no capítulo 3. O uso de um modelo como esse permite a melhor definição entre os blocos que compõem o *middleware*, especificando a relação entre eles de maneira explícita. Além disso, os mecanismos oferecidos pelo modelo de componentes auxiliam na construção de um sistema de configuração do *middleware* de acordo com regras pré-determinadas e podem permitir a definição de novas funcionalidades através de mecanismos de adaptação dinâmica, com o uso de interceptadores.

Mostramos, na seção 4.1, a estrutura de componentes escolhida com mais detalhe e na seção 4.2 o ciclo de vida normal de um cliente e de um servidor, através da seqüência de execução e da interação entre os componentes.

4.1
Estrutura

Através da pesquisa em *middlewares* adaptáveis, escolhemos fazer a refatoração do OiL, criando uma arquitetura baseada em componentes através da separação das suas funcionalidades principais em blocos. De acordo com

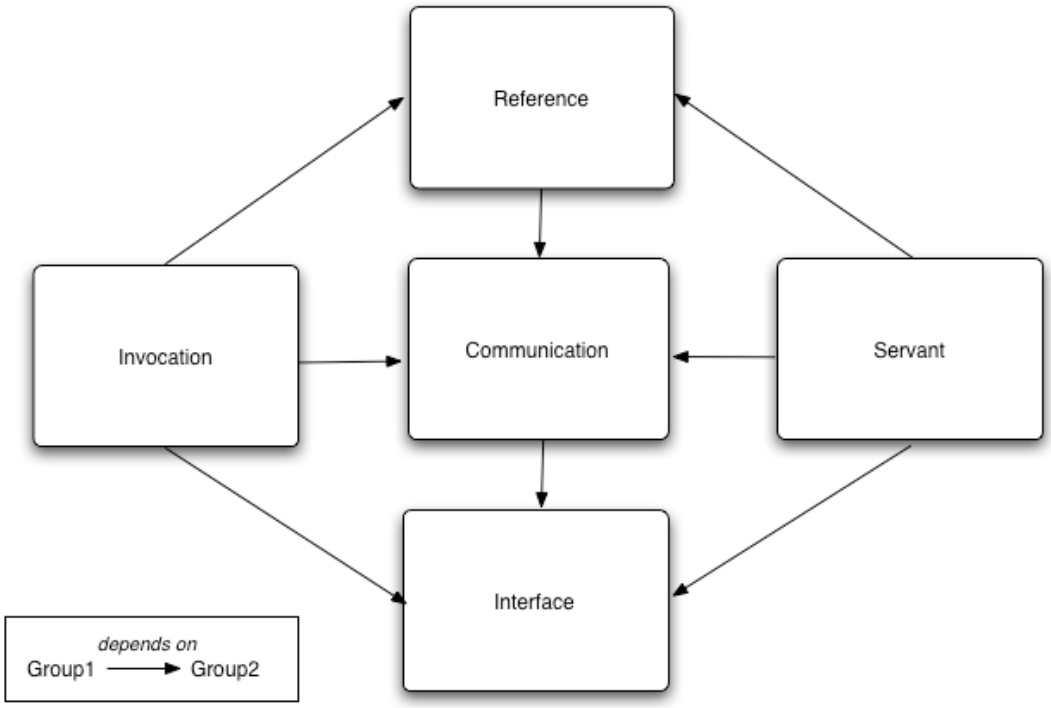

Figura 4.1: Arquitetura do OiL.

idéias apresentadas nos *middlewares* estudados, principalmente os baseados em componentes, dividimos o OiL em 5 grandes blocos funcionais: `Communication`, `Reference`, `Invocation`, `Servant` e `Interface`, com relações de dependência entre si mostradas na figura 4.1. Já a figura 4.2 mostra os componentes que formam cada um dos blocos e o apêndice B apresenta os tipos de componentes que formam a arquitetura do OiL usando a sintaxe do modelo de componentes do capítulo 3.

O bloco *Communication* apresenta as interfaces necessárias para fazer a comunicação de dados através de um transporte, como a codificação e a decodificação, a gerência de canais de transporte e da troca de mensagens entre objetos. Para a definição de como se conectar a objetos remotos existe o bloco *Reference*, que codifica e decodifica a referência para esses objetos. O bloco *Communication* usa essa referência para criar os canais de transporte necessários. Há também a divisão entre a implementação de objetos cliente e objetos servidor. A parte cliente de um objeto é representada pelo bloco *Invocation*, que faz a abstração da comunicação entre objetos, criando um objeto local que traduz as chamadas locais feitas a ele para chamadas remotas, usando para isso os blocos *Communication* e *Reference*. Já a parte servidor é representada pelo bloco *Servant*, que apresenta todas as funcionalidades de registrar um objeto local no *middleware* e, a partir desse registro, disponibilizá-lo para que objetos remotos possam invocar métodos fornecidos por ele, usando

para isso os blocos *Communication* e *Reference*. Finalmente, o bloco *Interface* funciona como um repositório de interfaces de objetos remotos, para protocolos onde essa interface precisa ser explícita. Assim, ao invocar um objeto remoto no bloco *Invocation* ou ao registrar um objeto no bloco *Servant*, o bloco *Interface* fica responsável por lidar com a interface desse objeto. Nas próximas seções, apresentamos em detalhe cada um dos blocos.

4.1.1 Communication

Este bloco é responsável pela parte de comunicação do *middleware*. Tem como componentes a parte de codificação e decodificação dos dados transmitidos pela rede, a criação e controle de canais de transporte de dados e o controle do fluxo de mensagens entre objetos remotos. Esse bloco, portanto, cuida de esconder dos outros blocos especificidades de um protocolo de comunicação.

Protocol

O componente `Protocol`, em seu tipo base, possui dois receptáculos. Um, `codec`, pressupõe a existência de um componente que seja capaz de codificar e decodificar *streams* de *bytes* para a transmissão. Já o `channels` declara a necessidade de se haver um componente que crie canais de transporte para o envio e recebimento dos *bytes* codificados e decodificados pelo `codec`. Além disso, depois de o processamento ter sido feito do lado remoto, a resposta deve ser enviada através do transporte. A implementação do `Protocol` não leva em consideração a política utilizada por `channels` na hora de criar e administrar os canais de transporte, ou seja, o `Protocol` pode implementar uma política de reutilização de canais de transporte independentemente da fábrica de canais ligada ao seu receptáculo `channels`.

O componente `Protocol` tem duas especializações: o `InvokeProtocol` e o `ListenProtocol`, para o lado cliente e servidor da implementação do *middleware*, respectivamente.

InvokeProtocol

O `InvokeProtocol` adiciona uma faceta, `invoker`, que exporta o método *sendrequest*. Esse método é usado por outros componentes para fazer a chamada remota, recebendo como parâmetro a referência para o objeto remoto a ser contactado, bem como os parâmetros da chamada em si. Como esse componente é uma especialização do `Protocol`, ele herda os receptáculos do componente original. Isso quer dizer que, durante a sua execução, este compo-

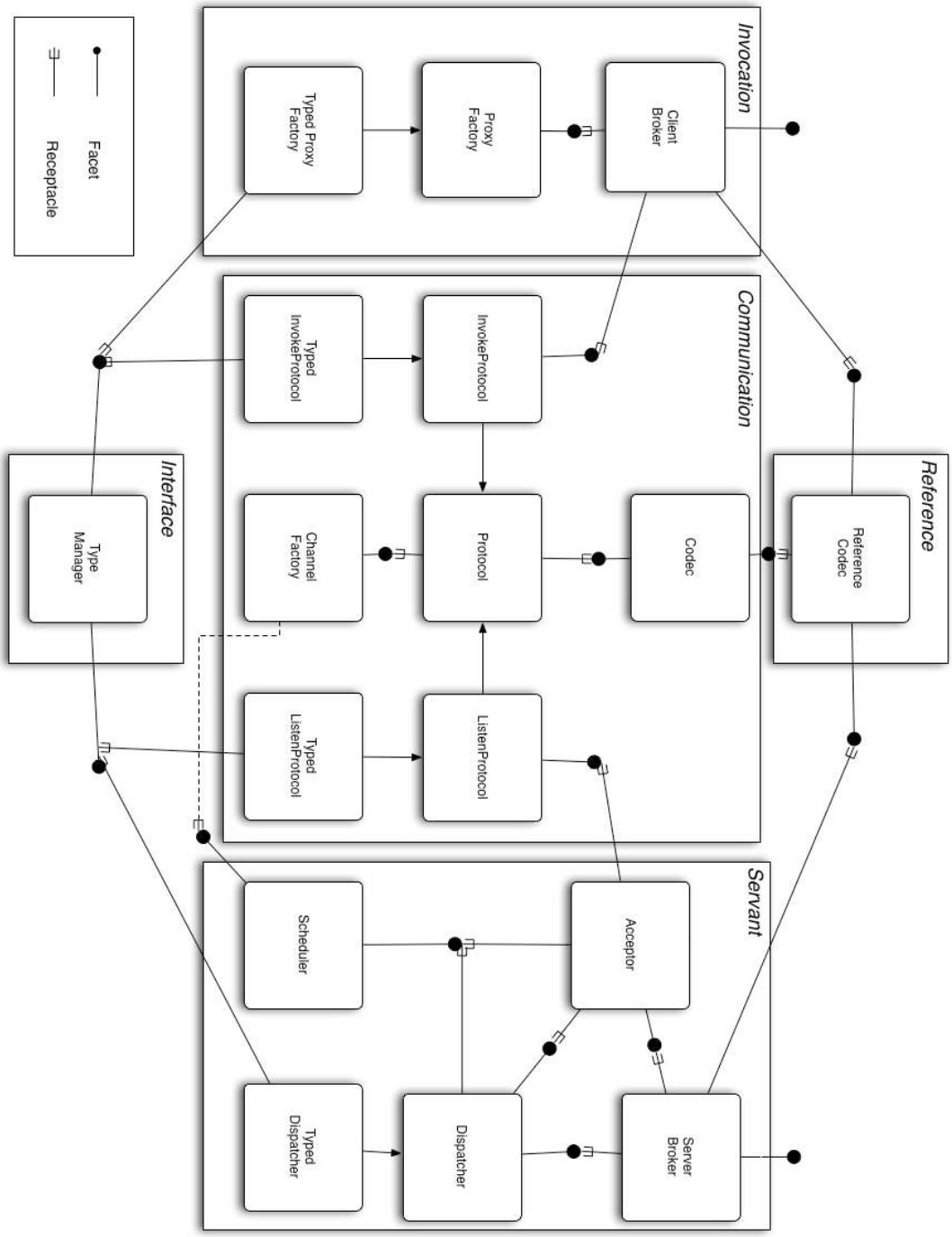

Figura 4.2: Diagrama detalhado de componentes do OiL.

nente utiliza a fábrica de codificadores e decodificadores e a fábrica de canais associados aos respectivos receptáculos.

A implementação de *sendrequest* retorna um objeto de *callback* com a implementação de um objeto, `ReplyObject`, que possui uma função chamada *result*. Ao invés de fazer a chamada síncrona, ou seja, o cliente enviar uma requisição e ficar bloqueado esperando uma resposta, escolhemos fazer uma separação entre esses dois eventos. Isso torna mais fácil a implementação de protocolos assíncronos. O `InvokeProtocol`, ao fazer a requisição remota, envia pelo canal de transporte a requisição e retorna imediatamente o controle para o componente que fez a chamada *sendrequest*, passando a ele o `ReplyObject`, com todas as informações necessárias para obter a resposta da requisição remota. Fica a cargo desse componente invocar a função *result* para receber a resposta de tal requisição.

TypedInvokeProtocol

Em protocolos em que há uma interface bem definida para as requisições remotas, isto é, há o número e tipo dos parâmetros e de valores de retorno, alguns controles devem ser adicionados à implementação do protocolo. No caso de CORBA, por exemplo, a codificação de dados para envio e a decodificação de dados de retorno é inteiramente ligada à interface IDL da função chamada. Por isso, a implementação da faceta `invoker` do componente `TypedInvokeProtocol` deve levar isso em conta.

Para auxiliar nesse trabalho, o `TypedInvokeProtocol` possui um receptáculo, `interfaces`, que oferece uma operação `lookup` para encontrar a interface correspondente à chamada remota sendo requisitada. Esse receptáculo é uma das ligações entre o bloco `Communications` e o bloco `Interface`, explicado mais adiante.

ListenProtocol

Assim como o `InvokeProtocol`, este componente adiciona uma faceta ao seu tipo base, `Protocol`, chamada `objects`, que tem uma função análoga à da faceta `invoker`, mas é relacionada ao lado servidor da implementação do protocolo. Essa faceta oferece duas funções: a *getchannel*, que recebe uma tabela com configurações e retorna um canal criado pelo receptáculo `channels`, através do qual o *middleware* vai receber chamadas remotas; e a *getrequest*, é a que recebe um canal, lê desse canal a requisição do protocolo correspondente, decodifica a mensagem usando a faceta `codec` e retorna imediatamente para o chamador, passando como valor de retorno um objeto `ReplyObject`. Este objeto, de forma análoga ao protocolo do lado do cliente, possui uma função

result, que recebe o resultado da chamada remota como parâmetro e o envia através do transporte. Essa separação entre o recebimento da requisição e a resposta a essa requisição também visa facilitar a implementação de protocolos assíncronos.

TypedListenProtocol

Novamente, em protocolos que possuem interfaces bem definidas, há a especialização `TypedListenProtocol` para o lado do servidor. Para obtenção da interface correspondente a um dado objeto remoto, cujo identificador vem incluído na mensagem recebida através do canal, há o receptáculo `interfaces`, que também se liga a componentes do bloco `Interfaces`, e que é usado por este componente através da função *lookup*.

ChannelFactory

O `ChannelFactory` é um componente responsável por criar canais de transporte. Cada canal é retornado através de um objeto `Channel`, que oferece uma API parecida com a do LuaSocket, implementando no mínimo as funções *send* e *receive*.

Duas políticas foram implementadas. A primeira implica em criar uma conexão para cada chamada, independentemente de existir uma já criada. A segunda, extensão da primeira, oferece um *cache* de conexões, para a sua reutilização caso já haja uma estabelecida entre dois *endpoints*. Dessa forma, a eficiência ganha na reutilização fica encapsulada de camadas mais altas da pilha de comunicação.

Codec

Funciona como uma fábrica de objetos responsáveis pela codificação e decodificação de dados a serem transmitidos pela rede. Oferece duas funções para obter um objeto codificador (*getEncoder*) e para obter o decodificador (*getDecoder*). Ambos objetos retornados por essas funções possuem uma API simples, com uma função para codificar (ou decodificar) cada tipo de dado existente na linguagem de descrição de tipos. A comunicação entre o componente `Protocol` e o objeto codificador/decodificador obtido do receptáculo `codec` depende do tipo de codificação esperado pelo protocolo. No caso de um protocolo como o IIOP de CORBA, os objetos do Codec devem implementar primitivas para a codificação de tipos declarados pela IDL de CORBA.

4.1.2 Invocation

Este bloco é responsável pela abstração de um *proxy* para o usuário final. Com essa abstração, as chamadas de métodos feitas a objetos retornados por esse componente são transparentemente traduzidas para chamadas remotas de procedimento. Esse bloco, através de uma referência para o objeto remoto, cria dinamicamente um objeto que serve como *proxy* local para tal objeto e usa ativamente o bloco `Communication` para a tradução dessa invocação para uma chamada remota.

ProxyFactory

É o componente que exporta uma faceta, a `proxies`, que representa uma fábrica de objetos *proxy*. A função *create* dessa faceta retorna um objeto *proxy* mediante a passagem de uma referência, já decodificada pelo bloco `Reference`, e do protocolo que vai ser usado. Esse *proxy* é então retornado à aplicação para ser usado como se fosse um objeto local. Cada chamada a esse objeto é capturada e enviada pela rede através do bloco `Communication` usando a função *sendrequest* apresentada anteriormente.

TypedProxyFactory

Uma extensão do componente `ProxyFactory` para *middlewares* em que há uma tipagem bem definida dos objetos, o `TypedProxyFactory` possui um receptáculo `interfaces`, ligado ao bloco `Interface`. Objetos *proxy* retornados por esse componente recebem a interface do objeto, obtida por esse receptáculo, para que, na invocação da chamada remota, algum tratamento possa ser efetuado no ambiente local. Por exemplo, no caso de CORBA, a interface auxilia na passagem de parâmetros para o bloco `Communication`, facilitando o trabalho de codificação dos parâmetros no momento da chamada remota.

ClientBroker

O `ClientBroker` é o ponto de contato da parte do cliente no OiL com o desenvolvedor final. Ele é conectado com o bloco `Communication`, para a definição do protocolo a ser utilizado na criação de objetos *proxy*, e ao bloco `Reference`, para decodificação de referências remotas, que são posteriormente usadas pelo `ProxyFactory`.

Este componente possui uma faceta, `proxies`, que recebe a referência do objeto remoto codificada e o nome da interface, se houver. Através dos três receptáculos que ele possui, um objeto *proxy* é criado: o primeiro, `reference`,

é usado para decodificar a referência, obtendo uma tabela, cujo formato é conhecido pelo bloco `Communication`. O segundo, `protocol`, é usado para obter o protocolo utilizado. O terceiro é o `factory`, ligado exatamente ao `ProxyFactory`, que usa a informação retornada dos dois primeiros receptáculos para a geração do objeto *proxy*.

4.1.3 Reference

O bloco `Reference` é responsável pela codificação e decodificação de referências remotas para objetos. Cada tipo de referência conhecido pelo sistema possui uma implementação de um componente, `ReferenceCodec`, para esse bloco. Por exemplo, se o *middleware* está configurado para usar o protocolo CORBA, o `Reference` deve ter registrado nele métodos para codificar e decodificar IORs, que são o modelo de referências de CORBA. Portanto, a implementação de um protocolo, gerando um novo bloco `Communication`, pressupõe a criação da codificação e decodificação de referências para esse protocolo.

Em muitos casos, para codificar e decodificar referências, utiliza-se o *codec* disponível no componente `Codec`, do bloco `Communication`. Dessa forma, há uma relação de dependência entre esses dois blocos, como mostrado na figura 4.2.

4.1.4 Servant

Este módulo é responsável pela parte servidor do ORB, ou seja, a criação de *servants* que respondem a chamadas remotas. Utiliza-se, para isso, facetas exportadas pelo bloco `Communication` para a criação de portas de comunicação que esperam por conexões vindas de outros objetos. Ao aceitar essas conexões, o bloco `Servant` deve saber para onde dirigir as requisições, a partir de informações vindas nessa conexão, de forma a completar o fluxo de execução da chamada remota de procedimento.

Os componentes que formam este bloco são:

Acceptor

Funciona como uma porta de entrada para requisições externas ao *middleware*. O `Acceptor` tem o receptáculo `listener`, ao qual conectamos um componente do tipo `ListenProtocol` e um receptáculo `dispatcher`, onde existe um componente do tipo `Dispatcher`, que será descrito a seguir. O `listener` é usado pelo `Acceptor` para a criação de uma porta de comunicação

e para a obtenção de uma requisição. Depois disso, ele passa essa requisição ao `dispatcher`, que a executará.

A faceta exportada pelo `Acceptor`, `manager`, exporta funções tanto para a criação desta porta de comunicação, através de configurações que serão utilizadas pelo protocolo, quanto para gerir a execução do `Acceptor`, fazendo-o esperar por requisições vindas da porta associada a ele.

O recebimento de chamadas inclui a espera por atividades na porta de comunicação. Se o `Acceptor` obtiver essa requisição e executá-la diretamente, essa execução é serializada. Para obter requisições de forma concorrente, ou seja, receber mais de uma requisição simultaneamente, é necessário a utilização de um escalonador. Este componente, por isso, possui um receptáculo `tasks`, ao qual pode ser conectado um componente que exporta funções de criação de *threads* para lidar com cada uma das requisições recebidas.

O `Acceptor` usa uma funcionalidade do modelo de componentes que é a possibilidade de saber ou não se um componente está ligado em seu receptáculo. Caso exista um escalonador conectado em sua porta `threads`, o `Acceptor` a utiliza. Senão, o recebimento de chamadas e o envio das mesmas para o `dispatcher` é feito de forma serializada. O componente escalonador é um pouco mais detalhado posteriormente.

Dispatcher

O `Dispatcher` é o componente que guarda referências para os objetos que serão exportados através do *middleware*, isto é, os *servants* criados do lado do servidor. Cada objeto é registrado através da faceta `registry` e, com isso, recebe um identificador único. Esse objeto é uma implementação em Lua das funções exportadas por esse objeto na sua interface, caso ela exista.

A outra faceta exportada, `dispatcher`, possui a operação *handle*, que recebe uma tabela com as informações da requisição. Essa tabela contém o identificador do objeto em questão, o nome da operação a ser executada nesse objeto e os parâmetros de entrada. O `Dispatcher`, nesse momento, recupera a referência para o objeto a partir do identificador e executa a operação. O resultado dessa operação é retornado ao chamador de *handle*.

O `Dispatcher` também possui um receptáculo ligado ao escalonador, da mesma forma que o `Acceptor`. Dessa maneira, cada execução de uma função dos objetos registrados neste componente pode ser efetuada em um novo *thread*.

TypedDispatcher

Da mesma maneira que nos outros blocos, em protocolos que usam um repositório de interfaces para a definição de tipos dos objetos, há uma especi-

alização de `Dispatcher`, com a adição de um receptáculo `interfaces` ligado ao bloco `Interface`. A sua função, do mesmo modo que nos outros blocos, é procurar a interface ligada ao objeto em questão para sua manipulação durante as chamadas remotas feitas a ele. No registro do objeto, a interface desse objeto é associada ao seu *servant*. Dessa maneira, ao receber uma requisição, este *servant* possui as informações de tipo do objeto. Na implementação deste componente para o protocolo CORBA, a interface serve para saber se a função a ser executada neste objeto é implementada pela própria infra-estrutura, i.e. uma função padrão de CORBA que pela especificação deveria estar presente em todos os objetos, ou se ela deve ser encaminhada ao objeto registrado no `Dispatcher`.

ServerBroker

De maneira similar ao `ClientBroker` do lado do cliente, o `ServerBroker` é o ponto de contato do lado do servidor para o desenvolvedor. Ele exporta duas facetas: `registry`, que oferece a operação de registrar objetos no servidor e a operação de obter a referência codificada para o *servant* criado; e `control`, que é a interface de controle do servidor, oferecendo a operação *run*, para a execução do mesmo. O registro de objetos é feito com o receptáculo `objectmap`, ligado a um componente do tipo `Dispatcher`.

O `ServerBroker` possui dois receptáculos de tabela associativa: o `ports` e o `reference`. Ao primeiro são associados componentes do tipo `Acceptor`, utilizando-se como chave da tabela o nome do protocolo usado nessa porta. Ao segundo são associados componentes do bloco `Reference`, usando-se a mesma chave de `ports`, relacionando o codificador da referência ao protocolo correspondente. Essa relação indireta entre os dois receptáculos visa a construção de servidores que respondem em mais de uma porta ao mesmo tempo, usando protocolos diferentes. O `ServerBroker` deve ser capaz de, a partir da sua configuração, criar a porta para cada protocolo e gerar as referências para os objetos registrados nele para cada protocolo diferente.

Scheduler

Para que o *middleware* receba mais de uma requisição simultânea na parte do servidor, utilizamos um escalonador baseado em corrotinas de Lua, usando o modelo de concorrência cooperativa (07)(56). O uso desse modelo facilita a programação concorrente, pois a mudança do fluxo de execução é feita em pontos bem conhecidos, minimizando a necessidade de o desenvolvedor se preocupar com problemas de sincronização entre *threads*.

O componente `Scheduler` engloba a implementação do escalonador, exportando duas facetas principais. A primeira, `tasks`, permite criar novos *threads*, passando a eles a função a ser executada, e geri-los, através de primitivas *suspend* e *resume*. A segunda faceta, `luasocket`, exporta operações da API de *socket*.

A implementação do escalonador leva em consideração a utilização de operações bloqueantes, tais como a leitura em um *socket* ou a espera por uma conexão, para a passagem de controle para outro *thread*. Por isso, o *middleware* deve utilizar as operações da API de *socket* exportadas pela faceta `luasocket`, ao invés de chamar diretamente as primitivas oferecidas pelo LuaSocket. Portanto, para se utilizar o escalonador, o `ChannelFactory`, do bloco `Communication`, deve se ligar à faceta `luasocket` e usá-la para criar os canais de transporte.

4.1.5 Interface

Este bloco fornece ao *middleware* as funcionalidades de um repositório de interfaces dos objetos remotos, servindo tanto para a descoberta de interfaces de objetos sendo utilizados como para a alimentação desse repositório com as interfaces de novos objetos.

Em protocolos que utilizam interfaces como uma forma de explicitar quais operações são oferecidas por objetos remotos, ao criar um novo *servant*, o identificador da interface e sua definição são passadas à função *update* da faceta `registry`, para a definição ficar disponível no repositório. Já a descoberta da interface é feita mediante a execução da função *lookup*, passando o identificador da interface.

4.2 Comportamento

Nas seções a seguir mostramos a seqüência de ações no lado do cliente e do servidor, para um melhor entendimento da interação entre os blocos e os componentes que o formam.

4.2.1 Ciclo de vida do cliente

Os passos de criação de um *proxy* (figura 4.3) são os seguintes:

1. O cliente pede ao componente `ClientBroker`, através da faceta `proxies`, um *proxy* para o objeto remoto a partir de uma referência codificada para ele.

2. O `ClientBroker` se comunica com a faceta `resolver` do `ReferenceCodec` ligado a ele, decodificando a referência remota. Internamente, o `ReferenceCodec` se comunica com o `Codec` conectado a ele para fazer essa decodificação.

3. De posse da referência decodificada, o `ClientBroker` a usa junto ao `ProxyFactory` para criação do *proxy*. Este é, então, retornado ao cliente, que o usará para fazer chamadas locais.

Já os passos da seqüência de execução de uma requisição remota do lado do cliente (figura 4.4) são os seguintes:

1. Quando o cliente realiza uma chamada sobre o objeto *proxy*, essa chamada é capturada por ele e transformada em uma chamada a *sendrequest*, função exportada pelo `Protocol`.

2. O `Protocol`, por sua vez, codifica a mensagem a ser enviada para o objeto remoto através de um objeto codificador, obtido do `Codec`.

3. Após isso, o `Protocol` usa o `ChannelFactory` para obter um canal de transporte para o objeto remoto, a partir da referência decodificada. Ele envia através desse canal a mensagem de requisição codificada. A decisão da criação de novos canais ou da reutilização dos mesmos quando a comunicação acontece com o mesmo servidor pode ser efetuada tanto no `Protocol` quanto no `ChannelFactory`. Esse aspecto está melhor definido na seção 4.3.

4. O objeto `ResultObject`, que possui a implementação da função *result*, usada no futuro para obter o resultado da chamada, é retornado ao *proxy*.

5. O *proxy*, quando quiser obter o resultado, pede-o ao objeto `ResultObject`, que internamente usa o canal de transporte obtido anteriormente para ler a mensagem de resposta.

6. Essa mensagem de resposta é decodificada por um objeto decodificador, também obtido do `Codec`.

7. Os resultados da chamada são recebidos pelo *proxy*, que os encaminha para o cliente.

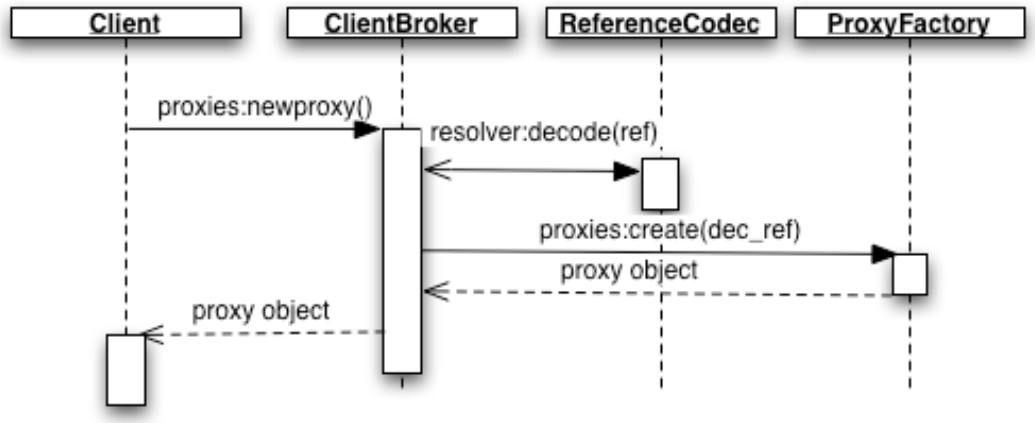

Figura 4.3: Criação de um *proxy*.

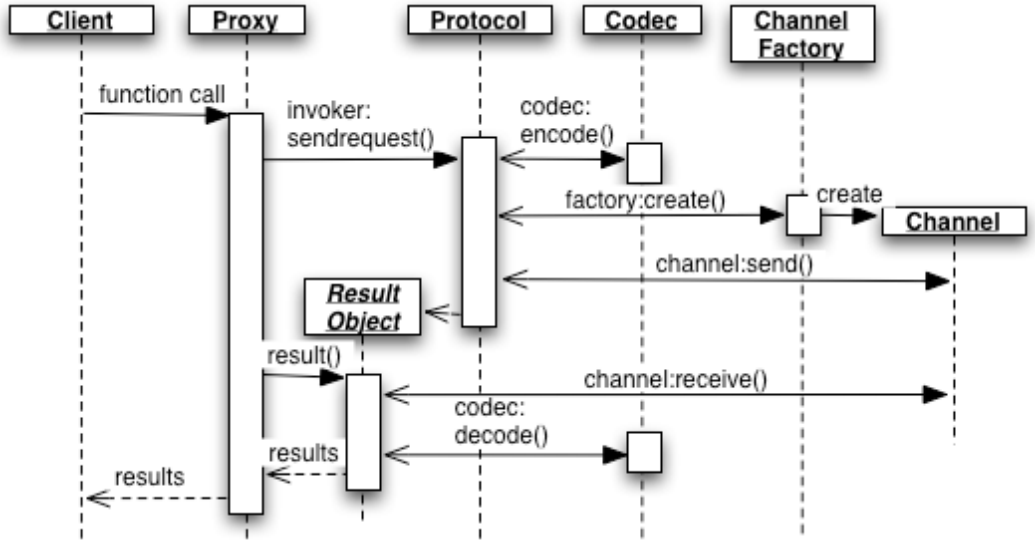

Figura 4.4: Seqüência de execução de uma chamada remota do lado do cliente.

4.2.2 Ciclo de vida do servidor

Os passos para registrar um objeto no *middleware* que fica aguardando chamadas remotas (figura 4.5) estão enumeradas abaixo:

1. Para registrar a implementação de um objeto no *middleware*, usa-se a faceta `registry` do componente `ServerBroker`, passando a implementação do objeto.

2. O `ServerBroker`, por sua vez, usa a função *register* do `Dispatcher` para registrar o objeto e obter o *servant*, que é retornado ao controle do desenvolvedor.

3. Para informar a clientes a forma de contactar esse objeto remoto, assumimos que existe no protocolo um suporte a referências codificadas que encapsulem esta informação. Essa referência para o *servant* criado é obtida passando o mesmo para o `ServerBroker` que, a partir do identificador do objeto presente no *servant*, obtém as informações do objeto no `Dispatcher`, através da função *getobjinfo*. Além disso, deve-se obter informações sobre a porta de comunicação associada a este servidor, a qual está esperando por requisições, através da execução de uma função *getportinfo* no `Acceptor` ligado a este `ServerBroker`. A arquitetura de componentes proposta para o OiL também inclui a possibilidade de haver mais de um `Acceptor` ligado a um processo servidor. Discutimos essa possibilidade com mais detalhe na seção 4.3.

4. Através do `ReferenceCodec` correspondente a cada porta disponível no OiL, o `ServerBroker` codifica a referência e a devolve ao desenvolvedor.

5. Finalmente, para iniciar a execução do OiL, o desenvolvedor executa a função *run* do `ServerBroker`, que chama *acceptall* do `Acceptor`. A porta ligada ao `Acceptor`, fica, então, esperando passivamente por uma requisição.

Depois de registrado o objeto, o *middleware* fica aguardando por chamadas remotas. A sequência de interação entre os blocos a partir da chegada de uma chamada (figura 4.6) é descrita a seguir:

1. Ao receber uma requisição, o *middleware* acorda o *Acceptor* ligado à porta correspondente, passando o canal de transporte a ele, que o repassa para o `Protocol`, através da função *getrequest*.

2. O `Protocol` lê a requisição do transporte e usa o `Codec` para decodificá-la. De posse das informações da requisição decodificadas, cria um objeto de requisição e a devolve ao `Acceptor`.

3. Esse objeto de requisição contém o identificador do objeto que deve executá-la, junto com o nome e os parâmetros da mesma. O `Acceptor` passa-o, então, ao `Dispatcher`, através da função *handle*.

4. O `Dispatcher` tem internamente a lista dos objetos registrados nele e, com o identificador do objeto, obtém a sua referência. Assim, ele pode chamar a função diretamente, ou criar um *thread* que irá executar essa função.

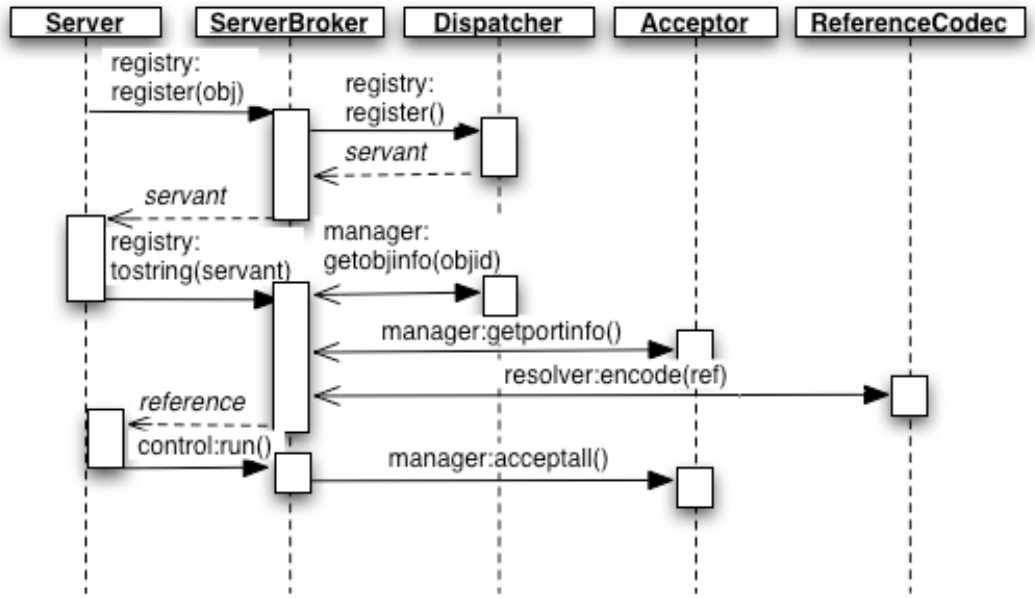

Figura 4.5: Criação de um *servant*.

5. Depois da execução local, o resultado é passado para a função, *result*, pertencente ao objeto de requisição, que havia sido criado pelo `Protocol`.

6. A implementação de *result*, de posse do resultado da requisição, codifica-o com o `Codec` e envia a mensagem formatada e codificada pelo canal de transporte, finalizando a chamada remota.

4.3 Considerações finais

O OiL, através da sua implementação seguindo a arquitetura de componentes descrita acima se torna um *middleware* configurável, ou seja, através da conexão dos componentes durante o seu carregamento podemos escolher quais os protocolos que serão utilizados ou mesmo se vamos utilizar uma política de escalonamento.

Construímos para isso um mecanismo simples de instanciação dos componentes e a conexão entre eles em tempo de carregamento do *middleware*, usando as primitivas apresentadas no capítulo 3. O desenvolvedor, em sua aplicação, escolhe uma das configurações pré-definidas, que já se encarregam de conectar os componentes de forma correta, ou pode, por exemplo, criar a sua própria configuração tendo como base uma das pré-existentes.

Na construção dos componentes acima, tivemos alguns problemas que não foram muito bem solucionados pela arquitetura apresentada. O primeiro problema é relacionado com a forma como foi componentizado o escalonador.

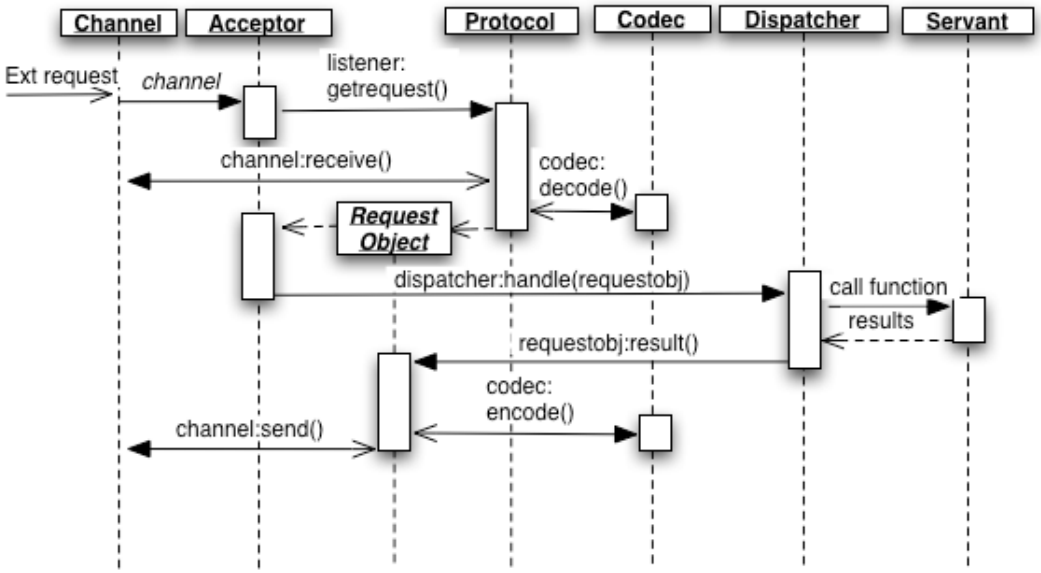

Figura 4.6: Seqüência de execução de uma chamada remota do lado do servidor.

Do lado do servidor, tanto o `Acceptor` quanto o `Dispatcher` são afetados na inclusão de um componente escalonador.

O `Acceptor`, para ficar aguardando por requisições, executa uma primitiva bloqueante na porta de comunicação. Quando essa porta recebe a requisição, o `Acceptor` acorda e envia essa requisição ao `Dispatcher`. Sem o escalonador, o `Acceptor` só voltará a esperar pela porta de comunicação depois do retorno do controle para ele. Com o uso do escalonador, esse período pode ser minimizado, mas o `Acceptor` para isso precisa saber da existência do componente que implementa a criação de *threads* para tal fim. Na proposta atual, o componente `Acceptor` possui dois comportamentos diferentes, de acordo com a existência ou não de um componente escalonador ligada à sua faceta `tasks`.

Já o `Dispatcher` tem um problema parecido, que foi resolvido na implementação de uma forma diferente, por ser mais difícil o isolamento em um só ponto do comportamento associado à existência ou não de um escalonador. Há uma versão do componente que executa diretamente a requisição recebida do `Acceptor`, mesmo que essa requisição faça o `Dispatcher` bloquear, para o caso da não utilização do escalonador. Quando o escalonador está presente, é utilizada uma versão do `Dispatcher` que usa as funções de criação de *threads* do escalonador para lidar de forma concorrente com cada uma das requisições que chegam a ele. A escolha do `Dispatcher` é efetuada na configuração do *middleware*, no carregamento do sistema.

Na parte do cliente, a presença do escalonador oferece algumas com-

plicações quando se adota a reutilização de conexões. Anteriormente mencionamos que a reutilização de conexões entre um cliente e um servidor pode ser configurada tanto no `ChannelFactory` quanto na implementação do `Protocol`. No primeiro caso, ao pedir ao `ChannelFactory` um canal de comunicação, dado um endereço de um servidor, ele pode escolher devolver um canal já existente, para minimizar os custos de criação de um novo. No entanto, problemas de acesso simultâneo podem aparecer quando se leva em consideração que mais de um *thread* diferente pode escrever no mesmo canal. Por isso, tornou-se mais interessante elevar o controle da reutilização de conexões para um nível mais alto. Por isso, na implementação do `InvokeProtocol` há, além de uma configuração de *cache* de conexões já existentes, uma comunicação implícita com o componente escalonador, caso ele exista. Apenas com essa comunicação o `InvokeProtocol` é capaz de impedir que dois *threads* diferentes tornem o fluxo de comunicação entre o cliente e o servidor inconsistente por misturar mensagens enviadas e recebidas pelo canal de transporte.

Como uma outra funcionalidade proposta na arquitetura do servidor, oferecemos também a possibilidade de haver a associação de mais de uma porta de comunicação a um `ServerBroker`, ou seja, mais de um `Acceptor` diferente associado a ele, ao mesmo tempo. Dessa forma, podemos usar a mesma implementação de um objeto para responder através de protocolos diferentes e, com isso, o desenvolvedor pode exportar um mesmo serviço para mais de um *middleware* diferente. Na seção 5.1.4 mostramos um exemplo fazendo uso dessa possibilidade. Entretanto, o fato de haver mais de uma porta ligada ao `ServerBroker` cria problemas no momento em que o desenvolvedor pede uma referência codificada para o objeto remoto, a fim de publicá-la para eventuais clientes que irão acessar o objeto. Dado que para cada porta existe uma referência diferente, codificada por um `ReferenceCodec` diferente, existe no sistema o mesmo número de portas e de codificadores de referências. O desenvolvedor do sistema que utiliza o *middleware* deve, portanto, fazer a associação explícita entre o protocolo desejado e a o codificador de referências que ele vai utilizar para codificar a referência ao objeto. O cliente que deseja se conectar a esse objeto deve, então, escolher a referência associada ao protocolo de comunicação desejado.

Outra consideração sobre a arquitetura proposta é a necessidade de haver implementações chamadas *typed*, tanto para o `ProxyFactory` quanto para o `Dispatcher`. Nos dois casos, a interface do objeto é obtida através de um componente do bloco `Interface` e associada tanto ao *proxy*, no caso do cliente, quanto ao *servant*, no caso do servidor. Os dois componentes que funcionam como fábricas na verdade não necessitam diretamente da interface do objeto,

mas apenas a usam na construção do *proxy* e *servant*. Um caminho possível seria a fatoração dessas duas fábricas, separando a fábrica de objetos e a lógica do objeto em si.

Como resultado dessa transformação do OiL em componentes, obtivemos um *middleware* adaptável na camada de distribuição, permitindo mudanças nos protocolos de comunicação envolvidos na construção do sistema distribuído. Além disso, o *middleware* resultante pode ser considerado, do ponto de vista do momento da adaptação, como um *middleware* configurável. A partir de arquivos de descrição, as partes que compõem o *middleware* podem ser escolhidas de forma condizente com as necessidades da aplicação. Na terceira classificação apresentada no capítulo 2, as técnicas utilizadas para adaptação, utilizamos para o OiL a organização de seus elementos em uma arquitetura baseada em componentes. Com isso, temos uma infraestrutura capaz de ser configurada de maneira simples.

Construímos, a partir do OiL baseado em componentes, alguns exemplos de configurações possíveis, que serão apresentados no próximo capítulo. Esses exemplos tentam aproveitar a separação de interesses da melhor maneira possível, fazendo a modificação da conexão de componentes dentro de cada um dos blocos `Communication`, `Invocation` e `Servant`. Pretendemos com isso demonstrar que a arquitetura é capaz de viabilizar e facilitar a configuração proposta no início deste capítulo e como o OiL se comporta com essas modificações na sua estrutura.